HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Telecommunications Software and Multimedia Laboratory

Ville Helin

Developments of Modern Multiplayer Games

Thesis is submitted for examination for the degree of Master of Science in Engineering

Supervisor: Professor Juha Tuominen

Inspector: M.Sc. Antti Nurminen

TEKNILLINEN KORKEAKOULU
DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Ville Helin

Työn nimi: **Developments of Modern Multiplayer Games**

Päivämäärä: 20.11.2003 **Sivumäärä: 76**

Osasto: Tietotekniikan osasto

Professuuri: T-111 Tietoliikenneohjelmistojen ja multimedian laboratorio

Työn valvoja: Professori Juha Tuominen

Työn ohjaaja: DI Antti Nurminen

Tiivistelmä

Työssä tutkittiin video- ja tietokonepelien nykytilaa tekniseltä kannalta, paino monen pelaajan verkkopeleillä. Samalla listattiin ja esiteltiin eri tekniikoita lisäämään pelien immersiivisyyttä, koska uusimmatkin pelit ovat teknisesti kohtalaisen yksinkertaisia. Tämän ohella esitellään yksi hieman valtavirrasta eroava tapa tehdä monen pelaajan verkkopeli.

Avainsanat: tietokonepelit, videopelit, MMORPG

HELSINKI UNIVERSITY OF TECHNOLOGY
ABSTRACT OF THE MASTER'S THESIS

Author:	Ville Helin	
Name of the Thesis:	**Developments of Modern Multiplayer Games**	
date:	20.11.2003	**Number of Pages: 76**
Faculty:	Department of Computer Science and Engineering	
Professorchip:	T-111 Telecommunications Software and Multimedia Laboratory	
Supervisor:	Professor Juha Tuominen	
Instructor:	M.Sc. Antti Nurminen	

Abstract

The objective of this Thesis was to list the techniques used in creating a modern multiplayer game, explain their limitations and suggest how the games could be improved. In addition to this the Thesis presents a different way (from the mainstream) to create a multiplayer game.

Keywords: computer games, videogames, MMORPG, networked virtual enviroments

Acknowledgements

This Thesis wouldn't have been possible without all those companies, which pollute the markets with uninspiring games. My biggest thanks to you. Without the high number of bad games I wouldn't feel this frustrated and actually write a Master's Thesis about it.

In addition I would like to thank all those who dare to experiment and create orginal games. Thanks to the people who worked hard to bring us "Gothic 1-2", "Silent Hill 1-3", "Thief 1-3", "Deus Ex 1-2", "Nethack", "Eternal Darkness", "Last Ninja 1-3", "Civilization 1-3" and many other brilliant and thus memorable examples of high quality games. Thanks to Shigeru Miyamoto, Hideo Kojima, Warren Spector, Andrew Braybrook, Matt Gray, Sid Meier and Matthew Smith. I wish I could remember and thank all you other talented artists as well, but I cannot. I have a bad memory.

I would also like to express my gratitude to Hugo Montesque, Pierre Sanchez, my parents and my brother, my friends and the dogs.

Helsinki, Finland
20.11.2003

Ville Helin

Contents

List of Tables

List of Algorithms

List of Abbreviations

AI	Artificial Intelligence
AIS	Artificial Intelligence Subsystem
BB	Bounding Box
BS	Bounding Sphere
BV	Bounding Volume
CFF	Critical Flicker Frequency
CS	Client-Server
DR	Dead Reckoning
fps	frames per second
FPS	First Person Shooter
FSM	Finite State Machine
GW	Game World
HMD	Head Mounted Display
LOD	Level-of-Detail
MOG	Multiplayer Online Game
MMOG	Massive Multiplayer Online Game
MMORPG	Massive Multiplayer Online Role Playing Game
MS	Mirrored Server
NPC	Non-Player Character
P2P	Peer-To-Peer
PBM	Play-By-Mail
PLG	Project Lecherous Gnomes (Chapter 5)
RPG	Role Playing Game
RTT	Round Trip Time
SDSL	Symmetric Digital Subscriber Line
SG	Server Grid
SMS	Short Message Service
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
VE	Virtual Enviroment

Chapter 1

Introduction

1.1 Background

Nowadays it is easy to find not only video and computer games based on movies, television series, books, comics and sports idols (e.g., "James Bond: Goldeneye" (1997), "Knight Rider" (2003), "Dune 2" (1992), "Jojo's Bizarre Adventure" (2002) and "Tony Hawk's Pro Skater 3" (2001)), but also movies and television series based on games (e.g., "Resident Evil" (2002) and "Pokemon" (1997)). It is not even uncommon that people sell (or sold in the case of "EverQuest" (1999), before its publisher Sony banned the trading [2]) virtual property obtained via playing multiplayer games in real world marketplaces. It would be a miracle if your part of the town didn't have a shop specializing in video games.

In the year 2000 about 60% of the population of the USA, around 145 million people, played video games on a regular basis [3]. According to ELSPA [4] the world market for video and computer games will grow to \$18.5bn this year, up from \$16.9bn in the year 2002. Video and computer games have evolved from A.S. Douglas' graphical computer version of Tic-Tac-Toe introduced in 1952 [5] to a growing industry, which might one day globally bypass movies in sales and displace them as the most popular entertainment medium. Today making a computer or a video game costs 3-6 millions euros on average and takes 2-5 years from a big team [6] compared to the early 80's when the team consisted of one to two persons and the project might have lasted only six months. While movies are linear and static, the games can be nonlinear and most of all they are interactive, which distinguishes them from the other media [1]. This technically makes games the superset of movies.

Games have the technical advantage, but only the time will tell how well it is exploited.

One and two player games have been the dominant game formats for long due to the relatively easiness of implementation. One might argue that the customers' high demand for such games has been the real reason for their popularity, but that cannot be. Only the recent advances in telecommunications technology have made massive multiplayer online games (MMOG) even possible, where tens of thousands of players can simultaneously roam in the same virtual world and interact with each other. In the Republic of Korea there are around 21 broadband subscribers for every 100 inhabitants [7]. (the number in the USA is only around 7), and playing alone is already considered to be a weird thing to do. Even though South Korean people have a stronger sense of community than e.g., the people in the USA, they might be showing the rest of the world the future direction of the gaming [8].

During the years there have been many discussions and studies about the effects video games have on people. Some argue that violent games increase hostility [9], some say the games will enhance the players' visual skills [10]. This Thesis will not make statements on the social and psychological effects of the games, but will focus on the technical side of modern multiplayer games.

1.2 Overview

In chapter 2 we list technical things that affect the playability of computer and video games. In chapter 3 we take a look on the history of multiplayer games, examine different multiplayer game architectures, the related problems, and spend a few words on different billing methods. Chapter 4 describes how the modern MMOG clients are constructed (graphics, artificial intelligence, etc.), what's wrong with them, technically, and how the technology might change in the future. The practical part of this Master's Thesis is in chapter 5 where the author's home brewn multiplayer game engine is dissected. Parts of the engine are much different from those used in commercial games. The final chapter 6 contains speculations about the future of gaming, and for those who are interested, appendix B shows few statistics about video game hardware in the USA as it tries to examine the hardware's effect on the sales.

Chapter 2

User experience in games

2.1 Theory

The amount of subjective illusion of presence, a feeling of being in a virtual enviroment (VE) instead of the real world, the VE creates in the user, is a common metric of VE's quality [22]. So high quality games are good at making the player to forget the reality. Unfortunately for the game manufacturers many different things affect the amount of presence the player experiences.

The better the hardware running the game the better are the chances that the player will immerse himself in the game [11]. For example, the effect of a 24 inch monitor is much more convincing than the effect of a 15 inch monitor. The narrower the field of view the less information about the game context it contains lowering the sense of presence [24], and this affects the player's ability to navigate and operate in the game world (GW) [25]. But even if the player had a state-of-the-art computer at his disposal the feeling of presence would instantly be shattered if the game paused for seconds to load a new level introducing a discontinuity in the GW [25], which is still the case with most modern computer and video games. Perhaps the best examples of a very discontinuous game are the games of the "Resident Evil" (1996) series. Here even outdoor areas are divided into sections connected with doors, and when the player moves from one into another an animation of an opening door fills the whole screen for few seconds. The player really has the time to open the door slowly even when chased by an army of zombies?

In appendix B we inspect how the relative level of the game console hardware affects its popularity, and will see that having the most powerful hardware doesn't mean it will sell well even though it could provide the players with the technically most advanced (i.e., most immersive) gaming experience. Many other things seem to affect the sales of the game systems as well.

2.1.1 Video

Humans start to see animation in successive images when they are changed about 8-12 times a second (frames per second (fps)). For the animation to start to look smooth the fps count should be around 24-30. But like Douglas Trumbull and his film projection system "Showscan" (60 fps, developed in the late 1970's), computer game players want more than 30 fps as image change rate directly affects the feeling of presence in the GW [15], [21].

Another crucial factor independent of the animation is the number of times the screen is refreshed per second. Cathode ray tube (CRT) displays (televisions and monitors) basically draw each frame by running a ray across the screen shooting electrons which, when hit the inner surface of the CRT tube, will emit light. So each refresh, when seen individually, would be a flash of light, but when the frequency of successive refreshs is high enough humans will perceive continuous light. Critical flicker frequency (CFF) is the term describing this limit, and it has been measured to be around 60Hz under nominal viewing conditions [16]. But studies show that in reality CFF is affected by illuminance of the room, luminance of the CRT [17], age [18] and gender [19] of the watcher, and many other things.

2.1.2 Virtual enviroments

If the outcome of the player's actions is not what the player anticipated, he might experience a decrease in the feeling of presence [12], [13]. If the game has fragile looking, small statues one should be able to move them and even smash them, especially if the emphasis in the game is on shooting and action. Being able to mofidy the GW's objects increases presence [11]. Also the novelty of the scenario has a positive effect on the feeling of presence, because people are more alert, more focused in new enviroments [23] and this way spend less time on real world events. So it should be a good idea to try to add new elements to the games. Letting the user to explore the GW also enhances the feeling of presence [11], [12], but it is very common to restrict the player's movements to not to let the player to go to areas, which the developers have not built due to various reasons. It is also unusually common to use thin wooden doors, wire-netting fence and a pile of small boxes to keep the bazooka wielding player inside the allowed perimeter. Many times these barriers are just invisible walls (e.g., "Freedom Fighters" (2003)).

2.1.3 Latency

End-to-end latency, the delay between the player's actions, e.g., pressing a button, and the moment when the outcome of the actions is displayed on the screen, has an effect on how the player experiences the GW. It is measured that low end-to-end latency results in greater feeling of presence than high latency [21].

If the game uses a network to transmit critical data during the gaming session, e.g., the position of the players in the GW, the network is part of the input/output chain of the game and thus affects the playability fundamentally. For example, if the player tries to pick up a bag of gold in the GW his client sends the server a request to do so. Only after the server's reply has arrived the player client knows if the player has the bag or not, as it could already have been taken by another player. If the answer would take a second to arrive, the player might become annoyed as he would eventually know his view to the GW wouldn't represent the game's actual status, and the high latency in interactions would make controlling the player character difficult.

2.1.4 Audio

The upper limit for human hearing is around 20kHz. According to the Nyquist-Shannon sampling theorem [20] we would need at least 40000 samples per second to be able to replay such frequencies, but this is no problem. Even the cheapest computers ship with integrated audio chips, which overexceed this basic requirement.

The audio requirements also depend on the type of the game. Most puzzle and card games work perfectly well without any kind of sounds as long as the game logic does not involve audio cues, but high adrenaline 3D action-adventure games like "System Shock 2" (1999) benefit a lot from 3D positioned high quality audio as it enables the player to use audio cues to locate his enemies before seeing them. With 2D audio the player might feel confused, e.g., "I can hear someone speaking, but where is he?". If the GW does not offer coherent and meaningful stimulus, e.g., the sound cues contradict with the visual information, the feeling of presence may abate [13].

2.2 Reality

The games in today's consumer markets concentrate on giving feedback to only two human senses, vision and hearing. Additionally few console games utilize a rumble pack integrated in some controllers. The rumble pack's only function is to shake the controller, but its purpose is questionable: If there is a big explosion in the GW and the controller jumps at the time of detonation, why do only the player's hands shake and not the rest of his body? Again, giving the players inconsistent information about the GW might not be a good idea [13].

The current trend is to offer improved graphics and audio game after game, but everything else is more or less the same as it was ten years ago. Games like "Max Payne 2" (2003), "Battlefield 1942" (2002) and "Star Wars Jedi Knight: Jedi Academy" (2003) are prime examples of games that have nice visuals, but indestructible enviroment. This is interesting, because already "Super Mario Bros." (1985) let the user to smash bricks to create holes in the walls, and using these holes the player could access hidden areas of the GW. Instead in "Max Payne 2" the player is forced to run through a tunnel disguised as buildings, streets and constructions yards. The only thing the player is allowed to do in this game is to march forward in the tube and shoot every bad guy he sees.

Yet not all games want to display realistic graphics. For example "Nethack" (under active development for over ten years) has ASCII character based graphics and still has a solid fanbase thanks to among other things the high level of interaction with the GW and non-linearity. In games like "Max Payne 2" your expectations are high, because of the fancy graphics, but your options are severely limited as you are only able move linearly towards the end of the game, which is just how the movies work. "Deus Ex" (2000) is an uncommon 3D game as it actually lets the player to accomplish tasks in different ways, like "Nethack". For example, to get pass a locked door one could pick its lock, blow it apart, find a key to it or just locate an alternative route leading to the area behind the door. One wasn't even forced to kill all the bad guys, only three, to finish the game so if one wanted to avoid violence he could sneak past most of the enemies if he had the skill. This freedom is missing from most of today's games, unfortunately. Also the player's options are often not numerous when interacting with artificial intelligence (AI) guided characters. In the worst cases, like "Golden Sun" (2001), a typical role playing game (RPG), most of the non-hostile non-player characters (NPCs) are capable of repeating few sentences over and over again, and nothing else. Also unfortunately for the players the existence of NPCs in the virtual enviroment does not enforce presence if one cannot interact with them [14].

Chapter 3

Multiplayer game architectures

Multiplayer games have evolved from simple games running on one computer to complex games that take years to develop and can have tens of thousands of separate computers around the world contributing into one huge gaming session. This chapter tries to explain the different ways of creating a multiplayer game, and the good and the bad sides of the various architectures.

3.1 Multiplayer games on one computer

It is possible to implement multiplayer games on one single machine, and this is in fact how the history of multiplayer games began. While most of the games require one input device for each player (like joysticks or mice) some work even by sharing one keyboard.

3.1.1 Shared screen

Most of the first computer games were designed for two players. Games like "Tennis for two" (1958), "Spacewar!" (1962) and "Pong" (1972) all featured two competing characters controlled by two human players. The display showed the whole playfield, e.g., a tennis court, and having a computer opponent was not always an option. The machines running these games were very primitive and creating an AI to play against would have required new hardware and a lot of effort. In many early games, which actually implemented computer guided artificial adversaries, one could see the limitations of the hardware as the AI characters moved e.g., from left to right to back to the starting very monotonously (like in "Jet Set Willy" (1983)), or almost randomly changed the direction after reaching a point in space with multiple exits (e.g., at a crossroads in a maze).

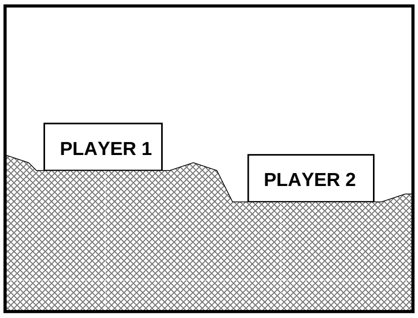

Figure 3.1: Shared screen

The shared screen multiplayer design is still use in today's games. Especially all fighting games, like “Super Smash Bros. Melee” (2001), use this technique to put from two to four player characters on the same screen.

Real-time shared screen games have one big technical limitation: The players cannot wander far away from each other as the view has to contain them all. In “Super Smash Bros. Melee” the side view camera zooms in and out according to the distance between the players, but in reality the camera cannot zoom very far away as otherwise the player characters would shrink into small dots. Turn based games don't have this limitation as one player controls the game at a time while the other player(s) just wait (e.g., “Artillery Duel” (1983)) for their turn.

The good thing in a shared screen system is that the implementation is relatively easy and the runtime computational requirements are low as one has to draw the view from only one camera (the transformations have to be done only once). Also the only view to the GW gets the maximum amount of onscreen pixels, which doesn't put any restrictions on the graphics like in the case with a split screen.

3.1.2 Split screen

Split screen games came later after the game hardware had become more powerful. Here the idea is to split the screen into multiple views, one for each player. Now the players could move far away from each other as each has its own view, which is independent from the other views. In the game “Pitstop II” (1984) the screen was horizontally split into two, each half showing the view behind the players' race cars.

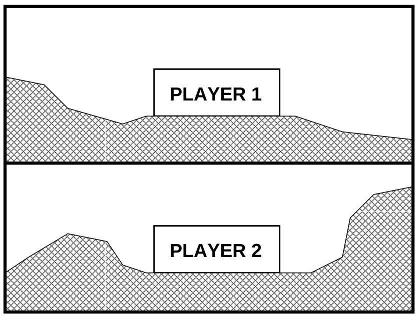

Figure 3.2: Split screen

Splitting the screen so that each player has its own part of the screen is computationally more expensive than sharing one view between the players. If the game uses 3D graphics and each view has its own camera, then in the case of a four player game we might do as much as four times the transformation calculations an one player version of the game would do, each frame. But as the number of players increase the size of each player's view decreases, and as the view gets smaller it is possible to drop the amount of detail in the graphics to speed up the rendering process. Still by splitting the screen each player has less pixels on their segment of the screen and the graphics will become coarser, which will decrease the playability of the game.

Many recent video game first person shooters (FPSs) use screen splitting to implement multiplayer gaming mode (like "Time Splitters 2" (2003)), but in general multiplayer games are moving into the Internet. The party games genre, which instances (like "Mario Party" (1998)) offer small and simple games for a lot of people staring at the same screen, might be the last one offering domestic shared and split screen multiplayer experiences without a connection to the outside world. Yet for quite some time one has been able to see shared screen multiplayer games on the Finnish television. Here the players (everyone with a GSM mobile phone) play the game by sending SMS (Short Message Service, 160 character long messages) messages to the game operator's service. This is in fact a modern version of Play-by-Mail (PBM) games, and it still has the same fundamental flaw all PBM games share: the user is forced to spend a lot of time on doing something that has nothing to do with the game itself, writing a text message in this case.

3.2 Multiplayer games on multiple computers

People with no friends who like to play multiplayer games have embraced the Internet with joy. Suddenly by plugging the computer into the Internet the only problem is to find suitable opponents from the masses of people waiting to being the game. By having also intelligent human characters in the GW instead of only simple AI algorithms will make the game much more interesting and varied. Even so, the possibility of playing with unknown people has created situations where some people attack other players in the game, without remorse, instead of monsters, which are the main targets, thus ruining the fun for the others. Fortunately for the many some multiplayer games enable individuals to host their own game sessions and invite only friends to play with.

3.2.1 Play-by-Mail

Play-by-Mail games go way back to the time before the Internet and computers. There might have been others, but at least Chess players were anxious PBM gamers. Here both Chess players had their own view to the game (two Chess boards with the same game state) and they sent their moves written into a letter and carried by the postal service.

The postal service acted only as a transmission channel so one could replace the Internet with postal service in all multiplayer games requiring multiple computers. The games would still work, but the delay in making a move would be unbearable as now the transmission would take days if not weeks instead of few seconds. Classifying PBM as a multiplayer game architecture is a bit misleading, but it is mentioned here as it has historical value.

The electronical version of PBM, Play-by-Email, has mainly superseded PBM. Finding statistics measuring the popularity of PBM games proved to be impossible, and as no PBM game has been mentioned in popular games magazines for years one can conclude that using electronical or paper letters to convey the changes in a game's state is today only the hobby of a few.

Play-via-SMS is the latest generation of PBM games. Some mobile phone operators offer text adventures, quiz games, etc., via SMS, but as it happened with the coming of the Internet, better protocols, better media and better hardware will soon replace SMS games with games where the message passing is integrated into the game software and is fully automatic. So instead of looking at a Chess board and writing the next move into a SMS message one could just move a pawn on screen and the system would transmit the information.

3.2.2 Peer-to-Peer

Peer-to-Peer (P2P) games, even though one could first think, are not the simplest of networked multiplayer games. In P2P all peers have a connection to each other. Each peer stores a local copy of the game state and updates it according to the command messages obtained from other peers. A minimal P2P game would consist of two computers, which was the most popular form of P2P games in the early years as it's easy to implement using modems with simple protocols.

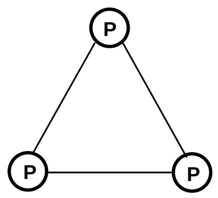

Figure 3.3: Peer-to-Peer

The good side of P2P architecture is that the peers communicate directly with each other, which should in theory keep the latency to the minimum. Maintaining consistency is a problem in P2P networks. A delayed or a lost message leads to consistency problems as the game state is duplicated across the peers and should be indentical across the network. This immediately raises an interesting question regarding the implementation of AI characters: If the game has n peers and m AI bots, how are they computed in a P2P architecture? Does each peer compute m/n AI bots? What if other players join the session after it has started? Or what if the game has simulated weather conditions. Which one of the peers decides the state of the weather?

3.2.3 Client-Server

Client-Server (CS) is quite a popular networked multiplayer game architecture. The reason for this is the relatively easiness of the implementation. The player clients communicate only with the server, which solely maintains the game state. The clients send and recieve updates, and the server distributes one client's effect to other clients.

In the CS model the server becomes quickly a bottleneck [26] as it has to process all the clients' messages and in addition to that usually compute e.g., AI characters. The server hardware must be very powerful to be able to handle numerous clients even though the server doesn't have to display graphics. It is easy to construct a game where one server is not enough e.g.,

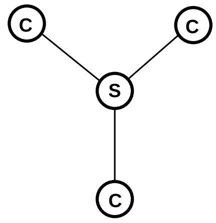

Figure 3.4: Client-Server

just add more clients until the server collapses. One can temporarily cure the situation by adding more servers to the same gaming session, and that will give us a Server Grid -model. One could also offload work from the server's CPU by introducing AI clients, but that in turn will require more network capabilities. The CS model suffers also from additional latency as each message is first sent to the server, which forwards the message or its effect to other players.

The good thing in CS architecture is that the one running the server can possibly e.g., expand the GW and fix bugs in the game logic without the need to touch the players' clients. Some CS games offer even automatic client updates when the client connects to the game server thus bothering the user as little as possible. This is not the case with single player games where the player has to find the updates in the Internet himself. Automating this process would be a technically trivial task, but still no single player game publisher does it.

3.2.4 Mirrored-Server

Mirrored-Server (MS) [27] architecture is almost the same as CS, but here the server has multiple duplicates (mirrors) spread over the Internet. The mirrors are connected to each other over a private, high-speed, low-latency multicast network, in a P2P fashion, dedicated solely to the game. Each client connects to the server providing the best service automatically.

The redundancy in the form of multiple servers makes the game network more tolerant to server failures than single server setups. Yet the servers are more complex as they require synchronization algorithms, and the need to synchronize the servers adds more latency to the game.

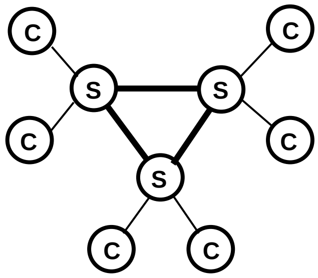

Figure 3.5: Mirrored-Server

3.2.5 Server Grid

In Server Grid (SG) architecture we again have multiple servers instead of one, but now the game state is shared between the grid's servers so that each server maintains a part of the virtual GW. For example, if we had a virtual house of four rooms we could set up a grid of four servers so that one server would take care of one room. When a player would move from room A to room B the player's client would transparently switch servers. In figure 3.6 the clients connect to a gateway server, which redirects the traffic to the correct game server depending on the player's position in the GW.

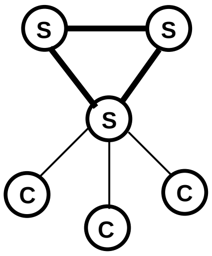

Figure 3.6: Server Grid

The good side of SG model is that one can assign more servers to more densely populated areas in the game. For example, a big city might require two servers while another server would alone maintain the game state in one rarely visited, but huge maze of tunnels. Dividing the GW properly will require heuristics about the GW and the habits of the players, and making this by hand can be difficult or even impossible. The latest SG systems are able to share the load dynamically, so if an army of players decides to meet at a special place in the GW the SG will react by changing the geometrical areas each server maintains on-the-fly, moving move servers to divide that special place, i.e., the concentration of dynamic objects.

The SG model is currently used in MMOGs [28] and is seen as the basis of many future networked multiplayer games. The same technology is used also by the banking industry [29].

3.2.6 Hybrid models

CS/SG and P2P hybrids come into the picture when the player clients need to exchange large amounts of data with each other, e.g., voice data. Circulating the voice communication data through the server is only waste of resources (unless the game server wants to analyze the audio, but not perhaps this decade) as the clients could transmit the data directly to those who hear it.

3.3 Scalability of the network architectures

Let's consider a case of n peers and n clients using UDP. In P2P each peer sends $n-1$ messages, one to each other peer, so we have a total of $n*(n-1)$ messages flowing through the network. In CS model each client sends one message to the server which relays the message or its effect to other $n-1$ clients. This means that the server works as multicast point for the messages, and $n*(n-1)$ messages will travel from the server to the clients. Client-Server model has less pressure on part of the transmission (client to server), but the server might collapse under all those messages it has to process and distribute. Server grids suffer also from this, but by adding more servers the grid will tolerate more players. In P2P networks the peers have to maintain a lot of simultaneous connections which might in turn kill the weak peers and overload the network. If only IP multicasting was widely available it would lower the architectures' bandwidth requirements considerably (especially P2P's).

See section 4.2.2 for a list of optimizations to speed up the implementation of all network models in a way not related to the underlying network architecture.

3.4 Common problems in networked games

The scalability or the lack of it is not the only problem in networked games. Delay (i.e. lag) introduced by the network and insufficient bandwidth are the two common sources of complaints, but they can mostly be fixed with new investments in hardware and network connections. But what is making the companies running game servers to close tens of thousands of players' accounts and access to the game is cheating [30], [31], [32], [33]. Cheating is also one reason why CS and SG network models are popular as there the servers are the only ones maintaining the game state. In P2P model the players' clients do it and are thus more vulnerable to hacking. The high number and arrogance of the cheaters has inspired even researchers to try to find solutions to the situation [34], [35].

3.4.1 Network latency and missing packets

The Internet was not designed for games. Currently it is a best-effort network where every packet of data is treated equally, routed individually, and it is not guarateed that the packets will reach their destination. It may create redundant copies of the packets, and even the order of the packets may change while they travel through the Internet. By using higher-level protocols like TCP/IP one can get rid off few of there problems, but at the same time reinforcing the delay in the transmission.

Turn based multiplayer games work well even in enviroments with high lag as they transmit a lot of data at a relatively low frequency. Let's consider a military strategy game: Each player takes his time planning and moving the units and ends his turn after he has used all his move credits. Real-time games, the majority of modern MMOGs, suffer highly from the lag, because they transmit small amount of data at a high frequencies. Let's consider a space game where every player controls a small, but fast spaceship: In ideal situation every client would know the moves every other client in the game made, each frame (i.e. each time the screen is redrawn), so every player would have a real-time view to the game's virtual world. See table 3.1 for examples of real world round trip times (RTT) between two machines, measured with the author's 2.3Mbps SDSL connection, and the corresponding frame rates the RTT values would support.

Still it is possible to make real-time games "work" even in high latency networks. Dead Reckoning (DR) technique tries to hide the delay, but can lead to other kinds of problems. Basically DR predicts the current status of the GW based on the previous data, if no current data is available (which is due to packet loss or lag). If the prediction is wrong, and quite often it is, players will experience jumps in the game state. For example consider a situation where player A stands still for a long time and then shoots player B, but the packet carrying the shooting information gets delayed in the network. Meanwhile the server dead-reckons that A is standing nicely behind B. Player B goes and buys a strawberry cake from a dealer D. Next player A's packet reaches the server and it will have to decide if it will tell A back that the bullet missed player B, tell B that he's dead and D that no deal was made or

Distance	Packet loss	RTT	adev	fps
5m (LAN)	0%	0.48ms	0.32ms	4166
1.5kms	0%	14.5ms	5.5ms	138
12kms	0%	10.7ms	6.2ms	186
1100kms	2%	43.3ms	3.6ms	64
2200kms	4%	117ms	8.1ms	17

Table 3.1: RTTs for a 128 byte UDP packet over the Internet. The underlying network, hardware and routes have not been analyzed. These values are meant to be examples only. The corresponding fps values are theoretical maximums for UDP given such RTT times.

return the whole game to the state before B fired his gun. A more common situation is one where player A's position is dead-reckoned to be few units in the wrong direction so when the correct information arrives the player A's image will make a visible and unrealistic jump from the wrong location to the correct location. Although it has its flaws DR is widely used in games and academic works, and is even part of High-Level Architecture [36] and Distributed Interactive Simulation [37] standards.

3.4.2 Cheating

If some players didn't want to cheat the world of online multiplayer games would indeed be nice as the game companies could focus on making the games better and not waste their resources on the fight against cheating. The situation has even spawned third party anti-cheat software [38] to help the players finding cheatless game servers (but are there any?).

Cheating usually begins by finding a patch to the game client in the Internet. Applying the patch will enable the player to for example autotarget, see through the walls, neutralize the effects of other players' weapons, remove distracting graphical effects (like screen shaking when hit by a bullet), and so on [39]. Cheat-proofing MMOGs is not easy, but is another essential requirement for success, because most of the people don't want to play an unfair game.

3.5 Billing in networked multiplayer games

With single player video and computer games the customer pays always only once, that is when he buys a copy of the game. With massive multiplayer online role playing games (MMORPGs) this is rarely the case. Usually the initial fee includes the game box containing client software and limited play time on the publisher's servers (players cannot run their own game servers). After the time is up the player has to pay more for additional time. Because the players pay for the time they play networked multiplayer games have the potential to become much more profitable than single player games, even though the initial fee for setting up a MMORPG is larger than creating and shipping a single player game [40] as MMORPGs require constant server administration and customer service. It is said that 90% of the work begins after the MMORPG is deployed [41]. Good scalability is also an essential requirement for a profitable MMORPG.

Monthly fees are not the only way to charge the players. One could also sell online play time instead of real world play time, or have a constant price on each play. Short games, e.g., shooting or puzzle games, are good examples of games where playing one round could cost a constant amount of money. As an extreme case one could have a turn based strategy game where each turn would cost a little.

In the future one could see MMORPGs where virtual items in the game cost real money. Some publishers might even provide mechanisms to exchange virtual money back to real money, which might in turn create a new occupation: people who get their living from a virtual world. Because of such prospects and the fact that tens of thousands of adults already spend more time playing online games than working some suggest [42] that MMORPGs may induce changes in the future societies.

Chapter 4

Client architectures

This chapter surveys many common techniques used in modern computer and video games. As we briefly go through them we also list their negative effects on the game mechanics, because one should note that some of the techniques impose severe restrictions on e.g., the player's freedom and the interactivity of the GW. The list is not meant to be exhaustive, and there are many better papers and books solely devoted to these algorithms, e.g., how to render complex scenes in real-time [43].

4.1 The client loop

See algorithm 4.1 for an example of the main loop in a typical real-time MOG client program. Some phases of the loop are often put behind interrupts or in separate threads [44]. For example, the playing of the audio might be triggered by an interrupt occurring every 1/60th of a second. Some clients might want to buffer the outgoing messages for example, for 100ms, to reduce the number of network operations.

Algorithm 4.2 shows an example of the main loop in a typical turn-based MOG client program. If you replace the phases 6-8 with AI computation the algorithm becomes the main loop of a typical turn-based single player game. The same can be done to algorithm 4.1 by replacing its phases 5 and 6.

4.2 Common optimizations

All modern games, be they single or multiplayer, use various optimizations and approximations to make the games playable on average consumer hard-

Algorithm 4.1: A typical client loop in a real-time game

1. Read the input from mouse and keyboard.
2. Act accordingly to the input (includes moving the player (players, if the game uses split or shared screen), computation of collision checks, etc.).
3. Redraw the screen.
4. Play some sounds.
5. Send all pending messages (e.g., position change notifications, pick up requests, etc.) to the server.
6. Process all the new messages from the server (changes in the game state).
7. Goto 1 unless the player wants to exit the game.

Algorithm 4.2: A typical client loop in a turn-based game

1. Read the input from mouse and keyboard.
2. Act accordingly to the input (includes moving the player, computation of collision checks, etc.).
3. Redraw the screen.
4. Play some sounds.
5. Goto 1 unless the player ends his turn.
6. Send all pending messages (e.g., position change notifications, pick up requests, etc.) to the server.
7. If the player has selected to exit the game, quit, otherwise play the waiting melody until the server gives the turn back to me.
8. Process the messages the server has sent me (changes in the game state)
9. Goto 1.

ware. A big part of programming a game is actually about choosing the suitable optimizations and implementing them. Some programmers even profile their ANSI C/C++ code (the most popular programming languages for writing a computer or a video game [44]), locate the functions where the CPU spends most of its time, and rewrite them in assembly language, at the final stage of the project. Doing so yields a linear speedup, but before spending time on optimizing the code it is essential that the used algorithms themselves are also optimized.

4.2.1 Minimizing network usage

It is crucial for the playability and scalability of a MOG to minimize the traffic from the client to the game server and vice versa [45]. All commercial MOGs come with CDs filled with game data, 3D geometry, textures, musics, and so on, and the games are constructed so that one can update these datasets with patches. What is moving from the client to the server and back is information describing how one should use the preinstalled data. This means that e.g., if one wanted to update the 3D model of a skeleton one should have to make a separate patch for it. If the game was designed so that the servers would always transfer the 3D geometry to the client when it needs that geometry (here a local cache on the client side would be helpful), then such patches had no meaning, and every client would have a real time view to the GW, and the game itself would take less storage space on the user's computer. The downside of such a fine idea is the increase in in-game bandwidth requirements. A hybrid version of this is a system where the client tells the server the earliest version number covering all the client's objects plus the version numbers for each object which has been updated after that. Afterwards, if the server notices that there is a newer version of an object available it sends the new version back to the client which updates its data storage accordingly. This method requires a little storage space on the server side and only a little of computing time when an object update is introduced to the players. There could also be separate patch servers which would only provide patches to the game. This way the updating process would not tax the game servers.

A trivial way to decrease the traffic is to use as little bits as possible in the messages. For example, instead of using ANSI C's "int" sized variables (four bytes) to hold the energy of an AI bot, which ranges from 0 to 100 in this example, one could use one byte as it has enough bits to hold all the different states. Another way to decrease the traffic is to compress the whole message data with e.g., LZW algorithm [46], if there is a lot of CPU time available at the both ends of the transmission.

4.2.2 Subdividing the space

If the design of the game allows, one can use spatial subdivision techniques to divide the GW into voxels. After the subdivision the only relevant voxels to the client are those that are near its focus point (e.g., the position of the player) in the GW. In networked games this means that the server needs to send the player only the state changes in such voxels. Without subdivision the server would either have to send the client all the updates in the game state (requires a lot of bandwidth and CPU time) or constantly compute the objects' positions in the GW and determine from that the relevant objects to a client (would still be computationally very heavy).

Note that all the subdivision techniques mentioned here, except BSP-trees, use axis-aligned grids to split the space.

Hybrid space subdivision is also possible. For example, one could use first a regular grid to subdivide the GW and then use an octree inside every voxel.

Bounding Volume Hierarchy

In bounding volume hierarchy [47] we surround objects with bounding volumes (BVs) and then try to recursively surround suitable clusters of BVs with even larger BVs. The hierarchy speeds up e.g., visibility checks as one can first determine if the parent BV is visible and if it is not, there is no need to process its children.

Constructing a good hierarchy is tricky, especially if there are moving objects in the GW. Unless one uses such a BV that encapsulates the whole subspace where an object can move, the program has to perhaps resize BVs as the objects move or then move the object from a BV to another BV, in real-time. Naturally one can use a different data structure for the moving objects and a bounding volume hierarchy for the static objects.

Regular Grid

Here we place a regular grid over the GW and use it to cut the geometry into voxels. Cutting will always increase the amount of polygons, and is suitable for static geometry only as one might not want to reconstruct and recut moving objects on-the-fly. Instead of cutting one can assign the geometry into such a voxel that encapsulates it the most. This means that the geometry comes partially out from the voxel it has been assigned to, but such a construction is fast to create and is suitable for moving objects to some extent. In the handling of such a subdivision one has to be conservative and examine all the voxels that are next to the voxels of interest as the subdivision is not exact.

Octree

Octree [48] is another hierarchical subdivision scheme. Here we enclose the GW, or a scene, with a voxel, and proceed to recursively subdivide each voxel into eight subvoxels of equal size if the voxel contains too much geometry.

As with the other hierarchical subdivisions, octrees are hardly suitable for holding dynamic geometry. A loose octree [49] is an octree where the voxels overlap, and we place each object into the smallest voxel enclosing it completely. Loose octree works well in scenes containing dynamic objects, but the looseness induces a small speed penalty as every segment of a space can belong to more than one voxel.

Recursive Grid

A recursive grid [50] is an octree where each voxel subdivision can generate any amount of subvoxels. Thanks to this ability recursive grids adapt better and result in a flatter hierarchy than original octrees when applied to the same scene.

Hierarchical Uniform Grid

Hierarchical uniform grids [51] are generated by first grouping adjacent objects of similar size. Next we use regular grids, where the voxel's size depends on the size of the objects in the cluster, to subdivide each cluster. After this we insert each grid into a higher level grid to construct the hierarchy.

BSP-tree

Many games including "Quake III: Arena" (1999), "Half-Life" (1998), "Unreal" (1998) and all those based on these engines, use Binary Space Partitioning (BSP) [52] trees for spatial subdivision. To construct a BSP-tree we first surround the scene (or GW) with a BV. Next we select one polygon from inside the BV and use its plane to divide the BV, and all the geometry inside, into two. We repeat this until all the polygons have been processed, no voxel contains more than the allowed number of polygons or the maximum tree depth is reached.

One can perform collision checks and hidden surface removal with BSP-trees relatively quickly, which contributes to their popularity, but because of the arbitrary voxels BSP-trees practically force the GW to be more or less static as rebuilding a BSP-tree takes time (the worst case is $O(n^2)$ where n is the number of polygons in the scene). This can be avoided by not subdividing such subspace where geometry is allowed to deform. By using only axis aligned subdivision planes such BSP-trees are relatively easy to construct.

4.2.3 Portals

Portals [53] are $n-1$ dimension polygons connecting n dimension cells (in 3D games $n=3$). Portals are used in architectural models where each room is one cell, or sometimes subdivided into smaller cells, and doors, windows and mirrors are portals. There is no similar analogy in outdoor landscapes, thus portals are basically used only when rendering indoor scenes. There are two types of portals: physical like doorways and virtual like mirrors, which cause camera transformations. Windows are special portals as one can see through them and also back to the current cell via the reflection on the glass.

With portals the visibility culling algorithm is as follows: First we find the cell where the camera is. Next we recursively list all the visible portals and cells beyond them.

Manually cells and portals are simple to create, but making an efficient algorithm to do the job automatically is more than challenging. Also portal rendering does not work optimally in situations where a small part of a cell is visible as cells are either completely rendered or not at all.

Portals, like BSP-trees, are quite popular in 3D games. For example, "Max Payne 2" and "Descent" (1994) use portals for visibility determination.

4.2.4 Frustum culling

One obvious optimization trick in a 3D game is to render only such geometry that resides inside the camera's view frustum, and not everything. Selecting such geometry and leaving away the rest is called frustum culling. This is usually done at run-time by calculating bounding boxes (BBs) or bouding spheres (BSs) for the 3D objects and testing if they are inside the view frustum, as testing the visibility of an object's BV is much faster than testing the object itself. The only downside in using a BV in the test is that even though the BV was visible the enclosed object could be invisible.

If the GW is spatially subdivided one can easily frustum test the voxels using BVs and that way optimize the rendering of the GW's geometry.

4.2.5 Ordered rendering

As the current hardware, with the exception of most mobile devices, has z-buffering [65] and some (ATI's R300+ graphics chips) implement partially even its hierarchical successor [66], one can optimize the drawing of 3D graphics in highly occluded scenes simply by rendering the geometry in front-to-back order. Even when done approximately, e.g., drawing the objects in this order regardless of the order of the geometry of an individual object, one can achieve a substantial speedup, especially if there is a high cost on putting a pixel on screen. Most of new games use multitexturing and pixel shaders, which make fully rendered pixels extremely expensive.

4.2.6 Potentially Visible Sets

If the GW is completely static, one can also construct Potentially Visible Sets (PVSs) [54] at the game creation time. Here we try to determine what geometry is visible for different camera positions and directions, and use that information at run-time to render the graphics. This also means that the space where the camera (i.e., the player) can move is strictly bounded.

Usually it is not possible to iterate all the possible stations for the camera. The common approach is to subdivide the GW into cells and approximate the exact set of visible geometry in each cell. Conservative approximations give PVSs which include geometry that is not visible, while aggressive methods give PVSs that lack some of the visible objects resulting in visual artifacts, but are naturally faster to render.

PVSs are often created by raytracing or drawing everything and collecting the visible geometry from the rendered image. Many racing and snowboarding games utilize PVSs as in them the player moves on a predefined track and the

4.2.7 Occlusion culling

Although z-buffering hastens the rendering of occluded scenes, all the geometry, even the invisible, inside the view frustum have to be submitted to the graphics pipeline operating the z-buffer. Advanved occlusion culling algorithms like hierarchical occlusion maps [55] decide at ealier stage the visibility of an object, groups of objects or voxels thus putting less burden on the rendering subsystem.

Instead of implementing any occlusion culling algorithms, many games use PVSs which already lack most of the occluded geometry and don't need real-time processing. But if one wants to have dynamic enviroments using PVSs is not an option.

4.2.8 Lightmaps

A lightmap is an image that contains information about the illumination of a surface, just like a texture map contains information about the structure of the surface. Lightmaps often have only eight bits of depth and smaller spatial resolution than the corresponding texture maps covering the same surface. When texturing the geometry one can reuse few template textures, e.g., rock, moss and wallpaper, to give the surfaces the details, and then enlighten each surface individually with its own, unique lightmap. The main idea when using lightmaps is to save graphics memory.

In games like "Max Payne 2" the lighting is precomputed using radiosity [56] algorithms, and the visual results are impressive. Unfortunately for the player even the lighting is then fixed, which makes the GW even less interactive. One could create multiple sets of prerendered lightmaps, e.g., one for such an occasion when a lamp is turned off and one for when it is turned on, but the storage requirements for the lightmaps would grow and the player would still not be able to move the light. Real-time generation of lightmaps is also possible, but CPU time consuming.

4.2.9 Billboards

Billboarding is about replacing a complex object in the scene with with its image often placed on a quad facing the camera. Billboards look convincing when their distance to the camera is long. For example, consider a situation where the player takes a nightly stroll in a small rural village in New England, and once in a while looks up to the Moon. It is quite evident that regardless of the position of the player inside the village the Moon looks always the same. So one can save a lot of CPU cycles by rendering a textured billboard instead of a Moon-like sphere consisting of thousands of polygons.

Some games go as far as replacing small vegetation like flowers and chunks of grass with billboards or statically aligned images, to speed up the rendering process. As the camera zooms into one of these impostors the player will soon notice the lack of the third dimension.

4.2.10 Particle systems

When games want to render e.g., fire, smoke and fountains, effects which display numerous small objects like water drops, they usually implement a particle system [57]. Here each particle is often a small data structure containing only its type, position, velocity, acceleration and life time. Also the forces affecting the particles are quite often much simpler than the forces affecting the larger objects, and games don't include particles in collision checks. The particles are visualized with low polygon objects, or even more commonly, with small billboards.

4.2.11 Level-of-Detail

There are numerous different Level-of-Detail (LOD) [58], [59], [60] algorithms for 3D graphics, but the basic idea is the same in all of them: when an object is far away from the camera and takes few pixels on screen it is not necessary to have the same detail in the object as when it is near the camera and contributes to many of the screen's pixels. The most used version of LOD discretisizes the object-to-camera distances into n separate classes, e.g., "near" and "far", and for each distance class there is a special version of the object. When we render the graphics at run-time we compute the distance class for every object each frame and draw the corresponding version of it. This will cause an object to pop when its classification changes, if the versions of the same object are different enough and there is only a couple of classes covering long distances. Some games even use billboards to represent the objects of the farthest class.

A system employing a discrete LOD algorithm requires only extra memory at run-time for the precomputed versions of the objects, which can be made automatically using various simplification algorithms [61], [62]. The same algorithms can be used inside the games to reduce the number of polygons in real-time if, for example, some events e.g., bombing holes to the ground, modify the GW's or objects' geometry and introduce lots of new polygons. A continuous LOD algorithm (also a well researched topic [63], [64]), on the other hand, needs only space for the original object and creates the simplified versions of it on-the-fly and thus needs lots of CPU time, but should also give visually better results as there are less bigger changes (less pops) in the geometry.

If all the LOD versions of the objects are loaded into the memory, making the switching between them instantaneous, one could make the graphics engine to decrease the displayed objects' LOD level by one every time the camera's speed exceeds a threshold to accelerate the rendering of the graphics. Many CAD programs implement a similar scheme by displaying a wireframe version of the object when it's moved and a textured, high quality version of the object when it stays stationary.

Other examples of LOD in modern games include rendering e.g., grass and pebbles only near the camera and fading them out as they are positioned farther away, and a LOD system for AI and animation. Many games spend CPU cycles on the instances of AI only if they are near the player, and if an AI bot is near the far end of the view frustum, it might just stand there getting no time from the CPU (e.g., "Baldur's Gate" (1998)). Naturally such a scheme does not work well for games where the AI is supposed to lead its own life. Animation LOD systems could e.g., skip animation frames or use lower degree interpolation between keyframes for objects which are far away from the camera and small on the screen.

4.2.12 Simplified collision checks

Not many games, if any, use the full visible geometry when computing the collision checks. For example, consider a box shaped house with very flat ornamets on the walls. Clearly it makes little sense to use the ornaments in collision checks as ornamentless, but otherwise identical walls would have almost the same effect. It is common to use simplified geometry in collision checks, and one can use the same simplification algorithms one used to create LOD objects, to create these simple models from the visible geometry.

Many games go as far as approximating the moving characters with boxes or ellipsoids in the case of collision checks. The player can quickly find out that there are invisible barriers around the characters, but in many cases it really does not matter.

One can also use BVs to optimize the selection of geometry in collision checks. For example, one could first test if the bounding volumes (sphere collisions are much faster to test than box collisions) of the objects collide before testing the enclosed polygons.

It is common to do the object transformations on the CPU, when dealing with collision checks. If we use simplified geometry in the collision checks we have to transform the visible versions of the objects as well when we need to display the graphics, so in the worst case we almost double the number of transformations each frame. Fortunately modern 3D video hardware is capable of executing the transformations, and very fast, so one can upload the visible geometry to the 3D card and let it handle them.

4.2.13 Simplified shadows

Another situation where the games commonly use simplified geometry is the rendering of shadows. For example, does the player see the shadow casted by a ring he is wearing? Many first games rendering shadows approximated the shadows the players casted with black circles. As long as the lightsources were high above the character's heads (e.g., the Sun at noon) the trick worked.

Nowadays it is common to use real-time computed shadows for moving characters, and precomputed, static shadows for the static GW geometry using lightmaps. Different types of algorithms exist for the shadow generation [67], [68], [69], [70], [71], and we are even seeing some functionality speeding up shadow rendering in the latest consumer 3D hardware (e.g., ATI Radeon 9800's shadow volume rendering acceleration).

4.2.14 Simplified physics

Most games, if they include any kind of physics simulation at all, use ridig body physics for the moving objects. Here no deformations are allowed, and the objects are replaced with their centers of mass in the computations. But during the recent years, with the increase in CPU power, one has seen few games implementing real time damage modelling for cars (e.g., "Burnout 2" (2002), "Mafia" (2002) and "Midnight Club 2" (2003)).

The 3D GWs are mostly static, and the author of this thesis knows only one RPG where it is possible to cut down trees ("Animal Crossing" (2002)). Causing damage to e.g., buildings is impossible in even today's RPGs.

4.2.15 Skydomes and skyboxes

Skydome is the top part of a sphere (in practice an approximation of a sphere, usually triangulated), centered on the player, and it is covered with textures of the sky. Some games use multiple layers of textures, for example, one for the stars and one for the clouds, and animate the cloud layer texture to create a crude illusion of natural sky. Skybox is a box shaped object used the same way.

4.3 Artificial intelligence

The current state of AI in computer and video games is miserable. The AI bots don't learn, they only exhibit hardcoded states [72] and mostly don't interact with each other. See algorithm 4.3 for an example of a typical deterministic finite state machine (FSM) enemy AI. There exist rare exceptions like "Black and White" (2001), which rebuilds decision trees, but only for the player's pet using its experiences as the data, and "Colin McRae Rally 2.0" (2001), where neural networks, taught by the developers, drive the computer opponents' rally cars. Also classic games like Chess, Go and many other strategy games use more complex mechanisms to breath life into the computer opponents, but adventure RPGs and 3D-shooters still suffer from simple AI.

Algorithm 4.3: A typical deterministic finite state machine enemy AI

1. Wait until I can see the player (start state). Then goto 2.
2. Run to the player. Then goto 3.
3. Hit the player with the best close range weapon available. If the player flees, goto 2. If the player dies goto 4, if I die goto 5, if I am low on hitpoints goto 7, else keep hitting (goto 3).
4. Victory for the AI. Goto 6.
5. Fall to the ground. Goto 6.
6. Delete this instance of the AI (end state).
7. Run away from the player. If not far enough goto 7, else goto 8.
8. Stop and heal myself. Goto 2.

Basically the AI bots know only about their surroundings via scripts made at the game creation time, so their ability to respond to changes in GW is very limited. For example, if the player destroys a colony of giant ants the AI bots in a village near by might still think the ant colony exists, which in turn might lead into decrease in the player's feeling of presence as the GW didn't work as the player anticipated [12], [13]. This also means that the scriptwriters must hardcode each one of the AI bots individually the information they need to know about the GW and other AI bots. Also it is a rare occasion to see the AI bots fight against each other. The AI seems to concentrate on monitoring only the player without doing a thing on its own. As an example, here is a sad, but typical scenario in most modern RPGs (taken from the game "Morrowind" (2002)): We have a small, but populated village. Many AI bots are standing in the market square, and some of them might walk randomly here and there, but not leave the market. The AI bots do not interact with each other, they just are there and wait that the player comes to speak with them. They have no needs nor do they know a thing about most of the others although they've lived in the village for years. The night falls, but the AI bots keep on standing and waiting. Next the player enters a hut in the village. He closes the door behind him, and brutally grabs a hankerchief from a table. Unfortunately for the player an old woman living in the hut witnessed this event, and so every AI bot in the village gets to know about this crime the moment it was committed. Telepathy? More likely bad design.

4.3.1 Triggers

Many games, especially single player games, use heavily triggers to guide the AI. A trigger can be an invisible polygon, which will signal the AI when the player steps on it. The most common case is that such a signal will result in the execution of a premade script that will, for example, make a monster jump to the player from behind the corner. “Half-Life” is a good example of a game based on triggers. After the monster has killed the player few times the player will learn to locate the trigger and avoid it in the future. Using triggers and scripts will also decrease the replayability of the game as the player can memorize the triggers all and thus the game becomes a series of deterministic events, much like a movie.

Instead of launching a script the triggered messages can carry useful information, which the AI will use when making decisions. For example, the gun the player carries could contain a trigger. Every time he would fire the gun the trigger would send a message saying that “a gun has been fired” to all those AI bots who would see or hear the gunshot, so the AI bots wouldn’t have to constantly examine their surroundings for such events, just wait for them to occur.

4.3.2 Pathfinding

There can be scenes in modern video games where hundreds of AI guided bots need to navigate simultaneously in the GW, so it may not be possible to render the view from every bot’s eyes and try to reason where it can go next. A very popular way to optimize the navigation is to place waypoints [73] into the accessible parts of the GW at the game creation time and build a navigation graph where each connection represents a valid path.

Some games store metadata into the waypoints. For example, in a first-person sneaking game (e.g., “Thief: The Dark Project” (1998)), waypoints could contain information about the illumination, the material of the floor (for how much noise it will emit when walked upon) and the presence of treasure chests and doors. By embedding the things that interest the AI into the waypoints the AI can be made simpler and more portable than if it acquired the data via other means.

A^* is the most popular search algorithm in games for finding paths between two points [49], [85]. If the GW is static one could get away with less by precomputing all the paths or at least some paths that may be used a lot. For example, if we had precomputed all the big paths between cities and our AI bot would want to go to one particular city C it would only have to find its way to the path leading to the city C or to the cities connected to the city C and then just follow that precomputed path. There are numerous other ways to find paths between two points and one could borrow various pathfinding algorithms from e.g., the field of robotics and implement them in games.

4.4 Content creation

Computer and video games are mostly data-driven. This means that the GW, characters, events and so on are handmade data constructed at the game creation time. This ensures that the game looks and plays the same every time, and the game creators put their effort into making the GW look and feel exactly the way they want, down to the smallest detail. The testing team is also happy as there is less to test than if the GW and the characters were not static and predefined. Unfortunately the down side is that the replay value gets smaller, because the players memorize the places, the objects, the events and the characters. As many of today's games take around 10-20 hours to finish (e.g., "Max Payne 2" took around six hours) the player might feel disappointed if there was no replay value in the game. Adding replay value to MMORPGs is essential, because if the player gets bored of the game due to the lack of variety and surprises he will not buy any more play time.

If the player is required to collect various objects the game could hide them into different places each play time. The game could also vary textures, AI attributes, even generate procedurally new levels. In "Nethack" most of the levels, but not all, are generated on-the-fly so no two game sessions are alike. Although some of the computer generated levels are plain and repetitive (due to the used algorithm) the player can start a new game and play quite a while without seeing a familiar level. It is easy to increase the replayability of a game, but even so only few games implement such methods while the vast majority offers identical scenarios play after play.

Chapter 5

Project Lecherous Gnomes

The main motivator for Project Lecherous Gnomes (PLG) was, and still is, the frustration the author of this Thesis feels when he plays many modern computer and video games. The games have nice graphics and audio, but the other technological aspects are still very primitive. The GWs are mostly static, and split into levels. Between the levels are irksome loading delays. For example, in the RPG "Morrowind", every house and hut are separate levels. When the player touches a door the game loads the interiors of the building and teleports the player inside. Even caves have doors in "Morrowind". And outdoors, once in a while, the game pauses for seconds, breaking any feeling of presence that is left, to load new segments of the GW. Rare exceptions include "Ultima IX: Ascension" (1999) and "Gothic" (2001) where the game loads new areas little by little so the GW looks and feels continuous, and even the buildings are in the same space with the landscape. Evidently continuous GWs can be done, but even so only a few games have them. Why is this? Bad design or incompetent programmers? Also the AI generally lacks in various departments as described in section 4.3. The other thing games lack is variety, so that every time the player starts a game from the beginning the story and the events repeat the same pattern.

The author of this Thesis thought that it cannot be very hard to address these problems. And it wasn't. PLG is a networked multiplayer game, but most of its technology can be used to create single player games running on a single machine. The purpose of presenting PLG in this Thesis is to show that creating continuous GWs and adapting game AI is possible, even from one person without any prior experience in game programming.

5.1 Concept

The technical concept of PLG includes three main parts. First of all, the game should be a MOG. Next we should have algorithms generating the content so that when the players get bored of the existing GWs they can create new ones by pressing a button. Also the AI, which should be adaptive, should be spread over the network like player clients.

The story of the game is not mentioned here as it is not relevant and necessary for describing the technical design of the game. Many might also find the story disturbing, and that it is as the name suggests.

5.2 Generators

Instead of replaying hand made content like most of the games PLG transfers the work from human artists to procedural algorithms. Here the idea is to generate a whole new GW deterministically from a single integer using pseudo random numbers so that by giving the same integer seed the generators will output the same GW every time. This means that instead of distributing hundreds of megabytes of premade data to the clients one can distribute only the seed integer and the clients will reconstruct the GW themselves before joining the game.

PLG contains two separate generators, one for the landscape and one for the buildings. To generate the whole GW we first generate the landscape and then let the building generator construct buildings into suitable spots reserved by the landscape generator.

One generator missing from PLG is the storyline generator. There could be an algorithm generating quests and events for the players, but to limit the amount of work it was not included in the project.

5.2.1 Landscape

The landscape generator in PLG uses diamond-square [74] algorithm to create random fractal terrains. The algorithm outputs a two dimensional height map, which is next converted to a 3D triangle mesh. After this the terrain is covered with handmade textures. Depending on the absolute elevation of a triangle a set of possible predefined ecosystems is obtained. By combining the information of triangle's relative elevation (e.g., valleys are wetter than ridges) and slope (e.g., not much grows on a steep slope) we select the final ecosystem for the triangle. More complex ecotope models [76] would give better results, but the current implementation is sufficient to show that

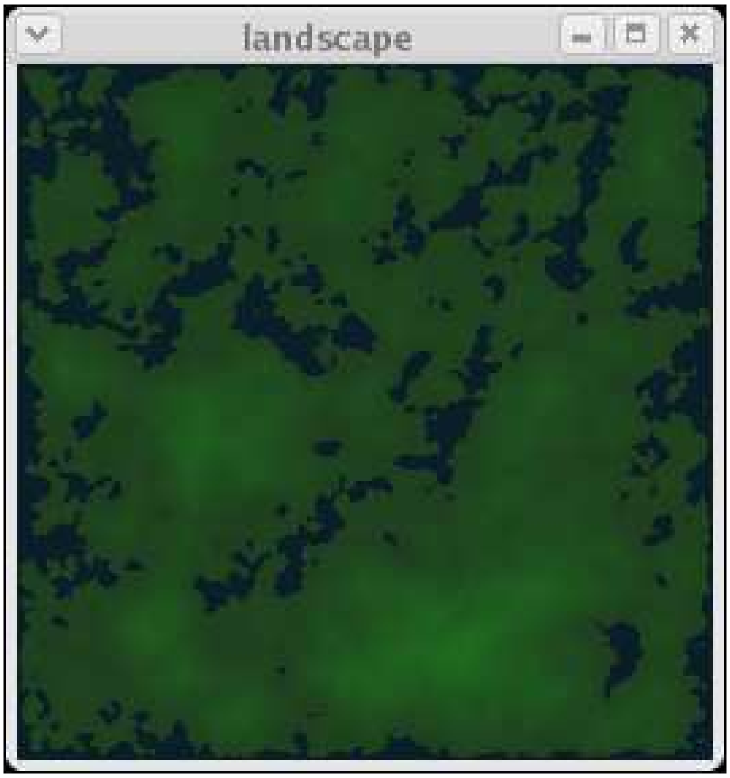

Figure 5.1: A terrain map generated using the diamond-square algorithm

the generator works and outputs useful data. The landscape generator also places trees, mushrooms and stones into the terrain (not the actual geometry, only coordinates and object ID numbers), based on the information extracted from the height map. All these objects are premade by hand, but one could integrate a vegetation generator to PLG, which would generate 3D trees and plants using e.g., Lindenmayer systems [77], to automatize the content creation even further.

The generated terrains look quite convincing and natural, but the players might find them boring as they lack any hint of the presence of an intelligent civilization, e.g., roads, mines, etc. Also visible borders between neighboring ecosystems plague the view.

5.2.2 House

"Soldier of Fortune II: Double Helix" (2002) included a random map generator, as an additional bonus to the real game, but while it felt like the terrain was procedurally generated the houses found on the generated maps seemed to be premade. PLG dares to be different and creates the 3D buildings room by room from premade elements (pieces of floor, ceiling, etc.).

The houses are generated initially in 2D space, but the geometry is reconstructed in 3D by replacing the 2D elements with 3D objects [78]. Algorithm 5.1 describes the whole process. What is missing are the room furnishing and facade generating algorithms, but they can be later added into the current generator. One should also somehow incorporate architectural knowledge into this process by perhaps modeling real life building designing, because by randomly inserting rooms on top of other rooms will give only more or less unrealistic houses.

Algorithm 5.1: The house generator in PLG

1. Clear the floor map (2D).
2. Insert random sized room rectangles to the floor map until there is no room for more.
3. Remove random number (0-n) of rooms.
4. AND the current floor map with the floor map of the floor below to make sure no room is above empty space.
5. Merge very small rooms with larger rooms.
6. Create doors between neighboring rooms until there are no unreachable rooms.
7. Use staircases to join the current floor to the floor below.
8. Move to the next floor and goto 1 until the desired number of floors have been created.
9. Go through all the floor maps and replace the 2D symbols with corresponding 3D objects to create the 3D model of the house.

The houses look a bit blocky, because they are constructed from predefined elements, and all the angles in the rooms are multiplies of 90^o. And as the houses are generated independently of each other and placed randomly on the map the result is anything but realistic. People don't build houses into random places, so the next logical step would be modeling the reasons behind building placement and embedding them to the landscape generator or head straight into city modeling [79].

```
    FLOOR 1        FLOOR 2
aaabbbbbcccc   aaaaabbbbbbb
aaabbbbbcccc   aaaaabbbbbbb
aaabbbbbcccc   aaaaabbbbbbb
dddbbbbbcccc   aaaaabbbbbbb
dddbbbbbeee.   aaaaabbbbbb.
ddSbbbbbeee.   aaaaacccccc.
dddbbbbbeee.   dddddcccccc.
fffffffggg..   dddddccccc..
fffffffggg..   dddddccccc..
fffffffggg..   .....ccccc..
```

Figure 5.2: A house of two floors generated using algorithm 5.1. The letters indicate the room where each piece of the floor belongs to, a dot denotes that the voxel is empty, and S symbolizes a staircase.

5.3 Spatial subdivision

After the landscape and houses have been created and merged into one, huge mesh, the data is cut into voxels using a regular grid explained in section 4.2.2. Each voxel is saved into its own file, and its coordinates are put into the file's name. If there is no geometry inside a voxel (e.g., a voxel above the ground in a valley) no file will be written, and the clients loading the voxels know that a missing voxel file is not an error.

5.4 On-demand loading

Every time the player moves the client checks that the voxels near the player are in the voxel cache. If some of them are missing the client loads one each frame into the memory providing the player a smooth transition into new areas. When the cache becomes full the client will free the most distant voxels to make room for the new ones. It is possible to teleport to distant places, and the only side-effect is that the cache gets flushed, meaning that right after the relocation there is a momentary decrease in fps. But at least there is no loading-screen halting the game for numerous seconds. The textures get the same treatment from the voxel loader: if a newly loaded voxel uses a texture, which is not in memory, it is loaded, and when the texture cache is full, old, unused textures are removed from the memory.

5.5 Visibility culling

The clients implement a two-stage frustum culling. The directions, where the player can look on the xy-plane, are divided into eight segments. First we find out the segments, which enclose the view frustum and discard the voxels that lie outside. After this we test the remaining voxels' BSs against the view frustum to get the set of possibly visible voxels.

After the frustum culling there is an optional occlusion culling phase (see algorithm 5.2). It is optional, because on some graphics hardware it actually slows down the program even though it will render less. The algorithm is pretty trivial and mentioned in many occlusion culling tutorials found in the Internet. The reason why it can fail to speed up the program is that rendering the BB of an object and getting back the result of its visibility can take more time than rendering the object itself. This seems quite obvious when the object consists only of few triangles, but still the slowdown can occur with complex objects due to the videocard's slow feedback system. Also waiting the videocard's answer will synchnonize the CPU and the graphics processing unit (GPU) on the videocard possibly stalling the CPU for a long time. Multiprocessingwise this can be turned into a better algorithm by letting the GPU tell it's answer when it's ready, but this will delay the answer and the CPU may ask for example the visibility of 20 objects before it starts to recieve answers meaning that the objects of this set will not contribute to the occluder data until after the delay. Evidently better algorithms should be used.

The clients use temporal coherence heuristics to decrease the number of occlusion tests. They assume that if an object is visible then it must be visible for the following 10 frames. Here the worst case is that the object becomes invisible after the first frame. Another heuristics is that if an object is invisible then it must be invisible next frame as well. This means that when objects become visible they may be displayed one frame too late. When the camera moves fast this causes the objects to pop into the screen, but with one frame error and high framerates this doesn't matter much. These two assumptions enable the engine to do at least 50% less occlusion tests than normally.

Algorithm 5.2: The trivial occlusion culling algorithm in PLG

1. Sort the objects by their distance to the camera.
2. Disable texturing, lighting and writes to frame and z-buffer.
3. Render the object's BB.
4. Read back the result saying how many pixels were rendered.
5. If the BB was visible enable texturing, lighting and writes to frame and z-buffer and render the object.
6. Goto 2 until all objects have been processed.

5.6 Collision checks

In PLG textures determine which polygons take part in collision checks. For example, trees' leaf textures are marked as non-collidable while brick wall texture is marked as collidable. When the collision checking routine has selected the voxels that need to be tested it filters away triangles using non-collidable textures. The friction information is also embedded into the texture structures and it is used in collisions to limit the sliding of the objects.

The implemented collision checking algorithm itself is quite common in games. Here we approximate the colliding object with an ellipsoid and test it against the GW's geometry. As an optimization we surround the space the object touches while it moves each frame with a bounding sphere and do not test such geometry, which doesn't intersect it.

5.7 Level-of-Detail

The LOD generator in PLG creates simplified versions of objects using edge collapse transformations [75]. Here we unify two adjacent vertices into a single vertex as long as the transformation doesn't exceed the given error threshold. By supplying the LOD generator different error thresholds we get objects of different resolutions, and using two predefined values a script generates two coarse versions of all PLG's objects for the in-game discrete LOD algorithm.

Currently the LOD engine doesn't handle landscape in PLG. That's because the LOD generator isn't suitable for optimizing such big meshes, and if one used it to simplify the voxels independently there would be cracks between them as the algorithm could move the outmost border vertices while collapsing the edges. Also it might be better to let the terrain generator create the simplified versions of the landscape from the 2D data as it should be much easier and faster than modifying 3D meshes.

5.8 Particle systems

The PLG's player clients implement two particle systems (described in section 4.2.10), one general purpose system taking care of point and polygon particle sources and one specialized system taking care of the clouds. Examples of point sources include burning torches and small water leaks in pipes, and examples of polygon sources include burning ground and fumes bursting out of large pipes.

The clouds in PLG move in three layers. To simulate motion parallax depth cue the topmost layer moves slower and the resolution of its clouds is smaller than compared with the bottommost layer. Each cloud is an unique billboard also generated with a variation of diamond-square algorithm. When a cloud has moved far enough from the player it will get a new, random starting point on the same layer. As the clouds fade towards black the farther they get from the player this relocation will be timed so it is not visible.

Experiments with this kind of a particle system showed that by increasing the amount of billboards while decreasing their size the clouds became more realistic, but at the same time rendering them took longer. Also the system initially recreated each cloud's billboard texture every time it was reintroduced to the layer, but as it was difficult to see the increase in variety as there were plenty of different clouds in the sky already, it was decided to drop this feature to save run-time CPU cycles.

The current cloud rendering system is just an enhanced 3D version of cloud engines seen in many old 2D games, and as long as the player cannot fly into the sky it fulfills its purpose moderately. 3D flight simulators and other applications, which need better clouds, use more advanced algorithms [80], [81], [82].

It should be noted that the particle systems in each client are not synchronized in any way, and for example, the same burning candle will look different in every client looking at it at the same time. But it should not matter at all, because the particle systems create only visual special effects and have no part in the actual gameplay.

5.9 Network model

PLG implements the permissible CS model, as described in section 3.2.3, with the addition of AI-clients. Here the server doesn't run the AI as we have separate client programs for that, and the idea is to offload the complex AI calculations to other machines at the expense of increased network traffic. The server sees the AI-clients as player clients, and there is only one interface for the clients. Permissible CS means that the clients always ask the server for a permission to do things, e.g., pick up a sword A or open a door B, and the server will later, if the client got permission, send all the clients nearby a message e.g., "player X picked up sword A". This will introduce latency in the game, but the system is easy to implement as without the need for DR scehemes there is no need for game state rollback algorithms either. To decrease the effect of the latency there is one exception: The players can move in the GW without the server's permission and the server doesn't verify that the moves are valid.

No security mechanisms have been included in the design of the network protocol. This is because the security system itself would provide material for numerous Master's Theses, and the author of this Thesis wanted to concentrate on the other aspects of multiplayer online games (MOGs).

PLG's network protocol works on TCP/IP. Many other MOGs use UDP, because it has less overhead than TCP, but unfortunately the communication becomes more complex as UDP doesn't resend missing packets or make sure the packets arrive at the destination in order.

5.9.1 Clients

Even though there are two kinds of clients in PLG, one for the players and one for the AI bots, both of them use the same interface to the server, and the server treats both types equally. Because of this it is possible to add novel AI clients to a running game on-the-fly, and if the players are dissatisfied with the existing AI technology they can write their own AI as the network protocol documents are included in the source code archive.

See algorithm 5.3 for the player client loop in PLG. At the startup the client connects to the server and acquires a starting position in the GW. After that it jumps to the loop. By replacing the phase 1 with AI computation the algorithm becomes the AI client loop.

Figure 5.3: View from the PLG's player client

5.9.2 Server

There is one server for each game session in PLG. It was decided that one server should be enough as the AI computation takes place on clients, not on the server. The server's task is only to maintain the GW's state and communicate with the clients. Algorithm 5.4 presents the PLG's server loop.

At the startup the server loads all the items (a list containing item's IDs and their positions in the GW) into its memory so it can control their usage. The server doesn't know about the GW's geometry, and the clients compute the collision checks themselves.

When a client connects to the server, the server first checks that the client's data has been generated with the same seed value than the server's data, and chooses a starting point in the GW for the player if the numbers match. After this the server monitors the client's movements in the GW. As the player moves the client sends the server its position, and how far in voxels it can see if the player adjusts the visibility settings. When the client sees a new voxel, or a voxel where the object information has changed (for every client the server timestamps the voxels it has sent them, and it also maintains universal timestamps for voxel changes), the server sends the client the object data.

Algorithm 5.3: The client loop in PLG

1. Read the input from mouse and keyboard.
2. Act accordingly to the input (includes moving the player, computation of collision checks, etc.).
3. Move the particles and clouds.
4. Redraw the screen.
5. Process all the new messages from the server (includes moving objects, creating and deleting particle systems, etc.).
6. Send all pending messages (e.g., position change notifications, pick up requests, etc.) to the server.
7. Goto 1 unless the player wants to exit the game.

Algorithm 5.4: The server loop in PLG

1. Check the main port for new clients. If there are any, let them join the game.
2. Go through all the connected clients and input the data they have sent me. Parse the messages and reply to the clients' requests by placing the replies into output buffers (one for each client).
3. Send all pending messages (flush the output buffers) to the clients.
4. Goto 1.

The server will also send few dummy object ID numbers to each client so that when the client needs to introduce an object into the GW (e.g., the player drops a book) it can give an ID number for the object without contacting the server. When the server notices that the client will soon run out of dummy IDs, it will replenish the client's dummy ID cache.

5.9.3 The protocol

See tables 5.2 and 5.3 for the messages the clients and the server send to each other. The messages are collected into a buffer, and each buffer starts with the message 's' indicating the size of the buffer so that when the buffer is sent the recieving end can read the first five bytes and know the total size of the incoming data.

Datatype	Meaning
i	integer, four bytes
d	double, eight bytes
f	float, four bytes
c	char, one byte
s	string, zero terminated

Table 5.1: Datatypes (big endian) in PLG's protocol

Message	Meaning
'd'.c, id.i	New dummy ID
'D'.c	The server has died
'I'.c, obj.i	Add object "obj" to your inventory
'N'.c, id.i, obj.i, x.d, y.d, z.d, ax.d, ay.d, az.d, name.s, co.c	New object
'n'.c, id.i, type.i, x.d, y.d, z.d, ax.d, ay.d, az.d	New particle system
'O'.c, id.i, x.d, y.d, z.d, ax.d, ay.d, az.d	Object's position has changed
'P'.c, id.i, x.d, y.d, z.d	Player's ID and position (only after login)
'p'.c, id.i, x.d, y.d, z.d, ax.d, ay.d, az.d	Particle system's position has changed
's'.c, size.i	The size of the messages
'S'.c, pla.s, mes.s	Player "pla" said "mes"
'T'.c, h.c, m.c, s.c	Current time
'X'.c, id.i	Delete object
'Y'.c, id.i, name.s	Object name change

Table 5.2: Messages from the PLG's server to the clients

Message	Meaning
'D'.c	The client has died
'L'.c, status.c	The status of the player's lantern
'M'.c, id.i	I'm missing object "id"
'm'.c, id.i	I'm missing particle system "id"
'n'.c, id.i, obj.i, x.d, y.d, z.d, ax.d, ay.d, az.d, name.s, co.c	New object
'N'.c, name.s	I want to change my name
'O'.c, id.i, x.d, y.d, z.d, ax.d, ay.d, az.d	Object's position has changed
'P'.c, id.i	I want to pick up this item
's'.c, size.i	The size of the messages
'S'.c, mes.s	Say "mes"
'V'.c, vis.c	Visibility settings have changed

Table 5.3: Messages from the PLG's clients to the server

5.10 Artificial intelligence

One of the targets of PLG was to create an adaptive AI to be used in RPGs. The AI should have the ability to learn its surroundings, see and remember the important actions the other creatures do, and most of all, exchange information with each other, remember the names of other creatures it has met and maintain a dynamic respect for each of them. These requirements were collected by inspecting how humans behave and construct social hierarchies, but only at a superficial level to keep the implementation task feasible. The author of this Thesis has no academic background in AI or psychology so what's presented here should be taken with a grain of salt. Even so the PLG's AI is much complex and believable than anything seen so far in commercial RPGs.

5.10.1 The structure

The starting point for the AI was borrowed from the simulation "The Sims" (2000), and its "Smart Terrain" model. Instead of hardcoding information about the GW into the AI, in "The Sims", each of GW's objects contain metadata (e.g., what it is, how it can be used, etc.) which the AI bots will realize when they see them. The players can insert new objects into the GW and the AI guided characters will adapt to the changes.

The AI in PLG is divided into two parts, the information processor subsystem (IPS) and the actual decision making AI subsystem (AIS), which operates only through the IPS. While the IPS is common to all different bots the AIS routines are not. This way it is possible create multiple, different AI profiles without the need to alter the GW or game logic.

PLG's AI model could also be used during the game creation process to initialize the AI characters and give them information about the places they live in, even if one didn't want to let the AI adjust its information during the game. This would enable the developer to drop AI bots into suitable places in the GW and let the system hardcode the local information automatically.

5.10.2 Information processor subsystem

All objects and creatures in the GW have an unique ID number, class ID and name. Every action, e.g., seeing, touching, hitting, etc., generates a message containing the subject, object, action and witness. These messages have many uses, and every AI bot witnessing the events will remember the messages, adjust their opinnions about the bots mentioned in them, and once in a while tell them to other AI bots they meet. This gossipping will be a vital source of information for many AI bots, because they will learn about the world that surrounds them from the rumors. For example, one bot can save the whole community by finding the only existing oasis in a large desert and telling about it to the others.

The IPS manages these messages and provides them to the AIS. This makes the IPS to act as a filter preprocessing the GW's information for the AIS. How the bots use the information accumulated by the IPS depends solely on the AI algorithms implemented in AIS.

The respect

The IPS is based on the concept of respect. Initially each AI bot knows only himself. Let's consider AI bot B_i. When he meets other AI bots or hears about them in messages (rumors) he will add them to his list. He evaluates every rumor by weighting the content by the respect he feels towards the teller. If he believes the rumor it will have an effect on him. Otherwise he will memorize the rumor, and if the same bot will tell it again, he can reject it once more, as that was his decision, but other bots are still allowed to try their luck. If the rumor tells that his friend B_a was hit by AI bot B_b, he will lower his respect towards B_b, but if B_a hits his enemy B_c, then he raises his respect towards B_a. IPS implements this behaviour with the following equations 5.1 and 5.2 for updating the respect B_i feels towards B_a ($r_{i,a}[n]$), who is the subject of the rumor, and B_b ($r_{i,b}[n]$), who is the object of the rumor. See the appendix A for the reasoning behind these equations.

$$r_{i,a}[n] = r_{i,a}[n-1] + w[p] * a_{subject}[p] * (r_{i,b}[n-1] - 0.5) * \sigma((r_{i,a}[n-1] - 0.5) * 10) \tag{5.1}$$

$$r_{i,b}[n] = r_{i,b}[n-1] + w[p] * a_{object}[p] * (r_{i,a}[n-1] - 0.5) * \sigma((r_{i,b}[n-1] - 0.5) * 10) \tag{5.2}$$

Here p is the event that took place (e.g., hitting), σ is the sigmoid function (5.4), and w is the effect of the event as described in equation 5.3. $r_{i,a}[n]$ and $r_{i,b}[n]$ are clamped to the range $[0, 1]$ after the updates.

$$w[p] = \begin{cases} a_{hear}[p] * a_{severity}[p] & \text{if one hears this rumor} \\ a_{see}[p] * a_{severity}[p] & \text{if one sees the event taking place} \end{cases} \tag{5.3}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.4}$$

In these equations $a_{subject}$ is the weight of the event p on the subject, a_{object} the weight on the object, a_{hear} the effect of hearing about the event in a rumor and a_{see} the effect of seeing the event. All these are in the range $[0, 1]$. $a_{severity}$ is the severity of the event, and it's in the range $[-1, 1]$. Here negative values mean that the event is negative (e.g., killing) and positive values mean that the event is positive (e.g., giving food).

By putting the events behind memory lookups we have succesfully taken the events outside the equations. This means that adding and removing an event will not require any changes to the IPS itself, only to the event lookup table offered to it via initialization routines. See table 5.4 for an example of an event table.

p	hear	see	subject	object	severity	event
0	0.9	1.0	1.0	0.0	-0.2	hit
1	1.0	1.0	1.0	0.0	-0.8	kill
2	0.0	0.0	0.0	0.0	0.0	see water
3	0.8	1.0	1.0	0.0	0.1	heal

Table 5.4: Example of an event table for the IPS

Additionally the IPS contains few rules for handling different situations (see appendix A for the complete functionality). For example, the AI bots discard all the duplicate rumors they hear so retelling the same rumor has no effect. A bot B_a may also hear a negation of a rumor he knew already from B_b. Here B_a will compute if he trusts the new rumor more than the old one, in which case he will negate the old rumor's effect (stored in the rumor structure) and experiences the new one, or if he trusts the old rumor more he will discard the new rumor and might even tell the old rumor back to B_b who also has to decide which one to trust. The rumor's credibility is the smaller of the respects towards the teller of the rumor and its alledged original teller (the witness). The IPS counts also experiences as rumors, so if bot B_a hears a rumor involving him although he has no recollection of such an event, he can start spreading a negation of it to make sure others will not get false information.

5.10.3 Artificial intelligence subsystem

The artificial intelligence subsystem is where the AI bots make the decisions based on the information the IPS provides. When an AI bot B_a sees another bot B_b he can get B_b's record from the IPS to examine the respect he feels towards B_b. B_a can also go through his list of rumors and select all those where B_b appears. How B_a should behave next depends completely on the AI algorithms B_a uses.

Algorithms 5.5 and 5.6 present two simple AI algorithms, which can be easily built using the IPS. Although they are FSMs they are able to adapt to the changes in the GW.

5.11 Future work

One AI character per one AI client seems like a waste of resources as each client needs to keep a part of the GW in its memory. It would be more efficient to be able to handle more than one AI character in an AI client, if the host machine had enough CPU cycles free to satisfy the needs of multiple AI characters, but this would actually pay off only if the AI characters spent some time near each other. But as players want large GWs, and the current architecture allows the user to run multiple clients on one computer, creating AI clients that can handle multiple AI bots might be a waste of time. Note that currently it is even possible to run all, the server, AI clients and a player client on one single machine, if it is powerful enough for the task.

Nevertheless there are many other things the author of this Thesis would want to add to PLG:

- The landscape and house generators need to be enhanced so that they will produce more varied and believable output.

- Fix the landscape generator to create LOD objects or program the clients to create them on-the-fly.

Algorithm 5.5: The thug bot (FSM). Note that states 1-8 monitor the bot's health and if it drops to zero a jump to state 9 is made.

1. If there is another bot B_a visible inside the view frustum I haven't seen for a while goto 2, else goto 8 (start state).
2. If I feel low respect towards B_a goto 4. Otherwise goto 3.
3. Go talk with B_a. Goto 1.
4. Attack B_a. If I'm gravely wounded goto 5, if B_b dies goto 7, else goto 4.
5. Run away from B_b. If not far enough goto 5, else goto 6.
6. Heal myself. Goto 1.
7. If wounded heal myself. Loot the body. Goto 1.
8. Go through the rumors and if I have heard rumors about locations of disrespected bots, walk a while towards the nearest alledged location, else walk to a random direction for a while. Goto 1.
9. Fall to the ground. Goto 10.
10. Delete this instance of the AI (end state).

- Add more generators (e.g., roads and vegetation).
- Add animation, audio and more items.
- Create more different AI bots. Experiment with reinforcement learning algorithms. Also some AI bots might have bad memory so they would forget rumors, and some could be chronic liars. Make the bots' personal attributes (intelligence, wisdom, constituion, etc.) affect their decisions.
- Add waypoint generation so simpler AI bots could be constructed.
- Make a benchmark program, which will decide if the client should use the current occlusion culling algorithm. Alternatively implement a better occlusion culling algorithm.
- Make the clients display grass and other small vegetation.

Algorithm 5.6: The informer bot (FSM). Note that states 1-6 monitor the bot's health and if it drops to zero a jump to state 7 is made.

1. If there is another bot B_a visible inside the view frustum I haven't seen for a while goto 2, else goto 6 (start state).
2. Create a rumor (or update an existing one) which tells that B_a was seen at this location. Goto 3.
3. If I feel low respect towards B_a goto 5. Otherwise goto 4.
4. Go talk with B_a. Goto 1.
5. Run away from B_a. Goto 1.
6. Walk to a random direction for a while. Goto 1.
7. Fall to the ground. Goto 8.
8. Delete this instance of the AI (end state).

Chapter 6

The future of gaming

6.1 Graphics

The next generation of graphics hardware will be able to render again much more triangles per second than the current hardware. At some point it may become possible to throw away texture maps and finally construct the 3D models completely from pure geometry. The making of increasingly complex models becomes also increasingly time consuming so it might be possible to see even companies specializing in 3D model creation in the near future.

Add-on discs, discs which add more content to a previously released stand-alone game, may increase their popularity among the publishers as they are built on existing game engines and use partially old data, so making an add-on disc is much faster than making a full game from the scratch. Already "The Sims" is a good example of a very successful game with numerous add-on discs. Also fewer companies will make their own graphics and physics engines as it is less risky to licence a well tested and maintained, commercial software modules off-the-shelf (existing, popular components include "Unreal" and "Quake III" game engines and "Havok" game physics kit). Even today it is possible to download new content from the Internet to few commercial games like "Tom Clancy's Splinter Cell" (2002) and "MechAssault" (2002), or buy Nintendo's "e-Reader" cardboard cards holding new games or adding new items and events to e.g., "Animal Crossing".

The resolution of the graphics pipeline will grow to at least 32 bits per color component from the currently dominant 8 bits per color component. All new consumer level display cards contain user programmable GPUs, which operate among other things on vertex, color and texture data and can create fancy real-time effects like motion blur, depth-of-field, per-pixel correct lighting and volumetric fire and fog. Same effects were previously prerendered into in-game movie clips, because the hardware was not powerful enough to calculate them on-the-fly. An increase in color resolution will be needed as calculating using 8 bits will give plenty of rounding errors and artifacts in the rendered images. Already some high-end consumer GPUs offer 32 bits resolution.

What will the increased graphical capabilities mean in the future games? The characters will have realistic hair, skin and personal faces (even run-off-the-mill enemies!), but the big question is that will that make the games better? The novelty is doomed to wear of quickly. Even on today's hardware the characters look quite realistic so whatever advances there will be in computer graphics they all will be small increments and perhaps irrelevant to the gameplay [83]. Soon the games will have to compete with each other in other technical areas than graphics. This has already happened with computer rendered movies. The visuals e.g., "Final Fantasy: The Spirit Within" (2001) and "Finding Nemo" (2003) present are so good there is not much room for improvements, so now the emphasis is on the plot and the characters.

One major benefit of having a powerful 3D card is that one can use high resolution screen modes. With the latest consumer hardware most of the games work smoothly even in 1600x1200 resolution. We can expect to have higher resolutions and bigger monitors in the future, and at some point in time the walls of the livingroom may become coated with display devices giving real-like 3D graphics via shutter glasses or something more advanced. Meanwhile a similar effect can be achieved with video projectors [84], but the concept is still too expensive for the consumer markets.

6.2 Interactivity

The interactivity in games is bound to increase. Currently most of the games have static enviroments where the player can interact only with a small subset of the visible property. For example, when entering a house in an RPG one might see a mighty sword hanging on the wall, but there was no way of taking that sword from its rack even though the rack had no lock. Yet one might find an indentical, but usable sword from one of the house's treasure chests. The GWs have been static for two reasons: It is easier for the programmers to make static than dynamic and interactive enviroments, and the maintenance of dynamic enviroments is computationally expensive while there are numerous optimizations available for handling static geometry. Yet "Red Faction" (2001) and "Red Faction II" (2002) let the player shoot real holes in the walls on a relatively slow consumer hardware ("Playstation 2"). It is interesting why other FPSs have not adopted this functionality.

In the future we might also see completely new input devices. All the modern joypads and joysticks are very similar to those bundled with the first video games. Nintendo experimented with "Zapper" (1985), a light-gun, and "Family Trainer" (1987), a dance mat, but although games using similar devices have been released even recently, their number is small. The latest new concepts come in the forms of Sony's "EyeToy: Play" (2003) and Konami's "Boktai: The Sun is in Your Hand" (2003). "EyeToy" includes a small camera which is used to digitize the player in real-time. The player will play the game's simple subgames as himself as his image appears on screen and interacts with the GW. "Boktai" (for "GameBoy Advance") has a sunlight sensor. The player uses real sunlight to charge his weapons in the game, although the weapons can overheat from too much light, and there are monsters that go hiding from the light.

6.3 Artificial intelligence

Game AI is another field which will improve in the future as there is lots of room for improvement, and big improvements can be done with relatively small efforts. The AI model described in section 5.10, even though highly heuristic and simple, is a major step towards a dynamic and realistic GW.

The game industry has mostly avoided complex AI algorithms (e.g., neural networks, genetic algorithms, etc.), because they are difficult to implement and especially difficult to debug [85], but this doesn't lead to anywhere as good implementations of complex AI models don't pop out of thin air. In the last resort, if the making of better AI for a MOG fails, one could always hire an army of people to play the roles of the NPCs. But not all NPCs need to be humans. Consider for example a war game: AI algorithms could guide the infantry units, but real people could act as the generals giving commands to the AI so every battle scenario might require only a couple of paid players to provide challenges to the customers.

6.4 Audio

Like graphics, audio has reached the point where it's good enough for most of the players, and all the future improvements will be small increments. The games already have 3D positioned audio, numerous effects and 16 bit resolution. The current trend is, like in graphics, to increase the resolution used when computing the effects. Today's best consumer audio cards use 24 bits for arithmetics (e.g., "Audigy 2" (2002)), and tomorrow it might rise to 32 bits. Support for 5.1 speaker setups is common, and 7.1 support is already available in the latest consumer hardware.

6.5 Content creation

Perhaps the biggest contributor to the long development cycles of computer and video games is the creation of the content. The GWs have become larger and the details plentier, but still most of the work is done by hand using CAD software and specialized authoring tools. Many artists have their own libraries of template graphics, but the task of filling a virtual world with detailed creatures, buildings and other objects remains huge.

So far the industry has avoided procedurally created content with few expections (e.g., "Nethack", "The Elder Scrolls: Daggerfall" (1996) and "Azure Dreams" (1998)). The reason for this is again the difficulty in constructing such generators, which would create useful data. This can also be seen by inspecting the few existing games implementing content creation algorithms. For example, a good 3D human head generator would in the context of an 80's soccer game output personal looking models of which many had mullets. The faces would have to be different enough so the players were able to associate names to them, but they still had to resemble real human faces. Even if the generator didn't function perfectly one could use its output as a starting point for the hand work, which is in fact what some people do. But by integrating the generators into the game the game's replay value would increase along with the number of happy customers.

It's apparent that more generators are needed in content creation to shorten the development time of the games. For example a versatile, procedural 3D building generator could be a very valuable asset in the near future.

6.6 Mobile games

The next generation of CPUs inside the handheld gaming devices will be capable of creating smooth 3D graphics, and hopefully the displays will evolve along with the rest of the hardware. Currently Nintendo's "GameBoy Advance" (2000) offers 240x160 pixels on its 2.9 inch display, and it is really difficult to see what is going on in the few and crude 3D games available for the platform. Nokia's "N-Gage" (2003) has 176x208 pixels on its little over

2.2 inch display, but has almost ten times more powerful CPU. “N-Gage” moves fewer and smaller pixels faster, but as chapter 2 described, bigger displays are required for increased sense of presence, not forgetting the effect on the ability to navigate in virtual worlds. Unfortunately for Nokia “N-Gage” flopped as a portable game console instantly after its launch.

Mobile devices need to be small to be portable so the device mounted display ultimately dictates the size of the machine and it cannot grow into the same dimensions one gets from the TV or desktop monitors. This limitation can be circumvented at least by using a head mounted display (HMD). Nintendo tried this with its “Virtual Boy” (1995). It displayed 384x224 pixels for each eye, but due to among other things its 50Hz refresh rate and the immaturity of the technology players experienced headaches, and the machine flopped soon after its launch. It has been a while since that. If the price on HMDs can be brought to consumer levels they might give a healthy boost to mobile game technology. In addition to HMDs, foldable electronic paper capable of displaying fluid animations [86] might also serve as a display device in the mobile video game consoles of the future.

So far most of the mobile games have been single player games, although mobile phones have built-in networking capabilities. It is also possible to link four “GameBoy” consoles together with link-cables to form a small, local network, but there are no pure multiplayer games available for the “GameBoy”, only bonus multiplayer subgames coming with single player games. “N-Gage” will be the first true mobile gaming device with the emphasis on networked games, and the near future will provide data on how well networked games work on mobile devices and how popular or unpopular they become.

Bibliography

[1] J. Steuer, "Defining virtual reality: Dimensions determining telepresence", *Journal of Communication*, 1992, vol. 42, no. 4, pp. 73-93

[2] G. Sandoval, "Sony to ban sale of online characters from its popular gaming sites", *http://news.com.com/2100-1017-239052.html*, 10th Apr, 2000

[3] Interactive Digital Software Association Report, "State of the industry report 2000-2001", *http://www.idsa.com/releases/SOTI2001.pdf*, 1st Feb, 2001

[4] Entertainment and Leisure Software Publishers Association Press Release, "Video games market growing faster than ever before", *http://www.elspa.com/about/pr/pr.asp?mode=view&t=1&id=368 &ref=home*, 11th Mar, 2003

[5] M. Bellis, "Computer and Video Game History", *http://inventors.about.com/library/inventors/blcomputer_videogames.htm*

[6] Marko Laitala, "Suomi kärkkyy pelien tekemisen suurmaaksi", *Tekniikka & Talous*, 25th Sep, 2003, no. 32

[7] International Telecommunication Union Press Release, "Number of global broadband subscribers grows 72% in 2002", *http://www.itu.int/newsroom/press_releases/2003/25.html*, 16th Sep, 2003

[8] D. Kosak, "Why is Korea the King of Multiplayer Gaming?", *http://www.gamespy.com/gdc2003/korean/*, 7th Mar, 2003

[9] R. Tamborini et al., "The Effects of Virtual Violent Video Games on Aggressive Thought and Behavior", *Paper presented at the 86th annual convention of the National Communication Association, Seattle, WA, USA*, 2000

[10] D. Bavelier and C.S. Green, "Action video game modifies visual selective attention", *http://www.bcs.rochester.edu/people/daphne/GreenandBavelier.pdf*, 29th May, 2003, Nature, vol. 423, pp. 534-537

[11] T. Sheridan, "Musings on Telepresence and Virtual Presence", *Presence: Teleoperators and Virtual Enviroments*, 1992, vol. 1, no. 1, pp. 120-125

[12] B. Witmer and M. Singer, "Measuring Presence in Virtual Enviroments: A Presence Questionnaire", *Presence: Teleoperators and Virtual Enviroments*, 1998, vol. 7, no. 3, pp. 225-240

[13] R. Held and N. Durlach, "Telepresence", *Presence: Teleoperators and Virtual Enviroments*, 1992, vol. 1, pp. 109-112

[14] T. Schubert, H. Regenbrecht and F. Friedmann, "Real and Illusory Interaction Enhance Presence in Virtual Enviroments", *Paper presented at the 3rd International Workshop on Presence, University of Delft, The Netherlands*, Mar 2002

[15] W. Barfield and C. Hendrix, "The Effect of Update Rate on the Sense of Presence within Virtual Enviroments", *Virtual Reality: Research, Development, and Application*, 1995, vol. 1, no. 1, pp. 3-16

[16] P. Webb (Ed.), "Bioastronautics Data Book", *Scientific and Technical Information Division, NASA, Washington D.C.*, 1964, SP-3006

[17] D. Bauer et al., "Frame repetition rate for flicker-free viewing of bright VDU screens", *Displays*, Jan, 1983, vol. 4, pp. 31-33

[18] C.B.Y. Kim and M.J. Mayer, "Foveal flicker sensitivity in healthy aging eyes", *Journal of the Optical Society of America*, 1994, vol. 11, issue 7

[19] C.A. Maxwell, "Flicker science and the consumer", *Information Display*, 1992, vol. 11, pp. 7-10

[20] C. Shannon, "Communication in the Presence of Noise", *Proceedings of Institute of Radio Engineers*, Jan, 1949, vol. 37, no. 1, pp. 10-21

[21] M. Meehan et al., "Effect of Latency on Presence in Stressful Virtual Environments", *IEEE Virtual Reality*, Mar, 2003, pp. 141-148

[22] M. Meehan et al., "Physiological Measures of Presence in Stressful Virtual Enviroments", *Proceedings of ACM SIGGRAPH*, 2002, pp. 645-652

[23] G. Fontaine, "The experience of a sense of presence in intercultural and international encounters", *Presence: Teleoperators and Virtual Enviroments*, 1992, vol. 1, no. 4, pp. 482-490

[24] J. Prothero and H. Hoffman, "Widening the field-of-view increases the sense of presence within immersive virtual enviroments", *Human Interface Technology Laboratory Technical Report R-95-4, Seattle, Washington: University of Washington, USA*, 1995

[25] M. McGreevy, "The Presence of Field Geologists in Mars-like Terrain", *Presence: Teleoperators and Virtual Enviroments*, 1992, vol. 1, no. 4, pp. 375-403

[26] A. Abdelkhalek, A. Bilas and A. Moshovos, "Behavior and Performance of Interactive Multi-player Game Servers", *Proceedings of ISPASS-2001*, Nov, 2001

[27] E. Cronin, B. Filstrup and A. Kurc, University of Michigan, USA, "A Distributed Multiplayer Game Server System", *UM EECS589 Course Project Report*, *http://www.eecs.umich.edu/~bfilstru/quakefinal.pdf*, 4th May, 2001

[28] Butterfly.net, "The Butterfly Grid", *http://www.butterfly.net/*, accessed 16th Jun, 2003

[29] D. Becker, "Sony taps grid computing for PS2 online", *http://zdnet.com.com/2100-1103-986195.html*, 27th Feb, 2003

[30] D. Becker, "Online gaming's cheating heart", *http://news.com.com/2100-1040-933822.html?tag=fd_lede*, Jun, 2002

[31] D. Becker, "Cheaters take profits out of online gaming", *http://zdnet.com.com/2100-1104-933853.html*, Jun, 2002

[32] BBC News, "Online cheaters face games ban", *http://news.bbc.co.uk/1/hi/technology/2221335.stm*, Aug, 2002

[33] B. Colayco, "Blizzard cracks down on Warcraft III, Diablo II cheaters", *http://www.gamespot.com/pc/strategy/warcraft3reignofchaos/news_6024304.html*, Apr, 2003

[34] N.E. Baughman and B.N. Levine, "Cheat-Proof Playout for Centralized and Distributed Online Games", *Proceedings of IEEE INFOCOM 2001*, Apr, 2001

[35] E. Cronin, B. Filstrup and S. Jamin, University of Michigan, USA, "Cheat-Proofing Dead Reckoned Multiplayer Games", *Proceedings of ADCOG 2003*, Jan, 2003

[36] "Standard for information technology, protocols for distributed interactive simulation", *Tech. Rep. ANSI/IEEE Std 1278-1993*, Mar, 1993

[37] F. Kuhl, R. Weatherly and J. Dahmann, "Creating Computer Simulation Systems: An Introduction to the High Level Architecture", *ISBN: 0130225118, Prentice Hall PTR, Upper Saddle River*, Oct, 1999

[38] Even Balance, Inc., "PunkBuster - Online Countermeasures", *http://www.punkbuster.com*, accessed 17th Jun, 2003

[39] M. Pritchard, "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It", *http://www.gamasutra.com/features/20000724/pritchard_01.htm*, 24th Jul, 2000

[40] P. Palumbo, "Online vs. Retail Game Title Economics", *http://www.gamasutra.com/features/business_and_legal/19980109/online_retail.htm*, 9th Jan, 1998

[41] J. Mulligan, "Online Gaming: Why Won't They Come?", *http://www.gamasutra.com/features/business_and_legal/19980227/online_gaming_why_intro.htm*, 27th Feb, 1998

[42] E. Castronova, "Virtual worlds: A first-hand account of market and society on the cyberian frontier", *CESInfo Working Paper No. 618*, Dec, 2001

[43] T. Möller and E. Haines, "Real-Time Rendering", *ISBN: 1568811012, AK Peters, Ltd.*, Jun, 1999

[44] A. Rollings and D. Morris, "Game Architecture and Design", *ASIN: 1576104257, the Coriolis Group*, Nov, 1999

[45] J. Begole and C. Shaffer, "Internet Based Real-Time Multiuser Simulation: Ppong", *Technical Report TR-97-01, Virginia Tech Department of Computer Science, USA*, Feb, 1997

[46] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, May, 1977, vol. 23, no. 3, pp. 337-343

[47] S. Rubin and W. Turner, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes", *Proceedings of ACM SIGGRAPH*, 1980, pp. 110-116

[48] A. Glassner, "Space subdivision for fast ray tracing", *IEEE Computer Graphics and Applications*, Oct, 1984, pp. 15-22

[49] M. DeLoura (Ed.), "Game Programming Gems", *ISBN: 1584500492, Charles River Media*, Aug, 2000

[50] D. Jevans and B. Wyvill, "Adaptive voxel subdivision for ray tracing", *Proceedings of Graphics Interface*, 1989, pp. 164-172

[51] F. Cazals, G. Drettakis and C. Puech, "Filtering, Clustering and Hierarchy Construction: a New Solution for Ray Tracing Very Complex Enviroments", *Computer Graphics Forum*, Aug, 1995, pp. 371-382

[52] H. Fuchs, Z. Kedem and B. Naylor, "On visible surface generation by a priori tree structures", *Proceedings of ACM SIGGRAPH*, 1980, pp. 124-133

[53] J. Airey, "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations", *PhD Thesis*, *University of North Carolina*, Jul, 1990

[54] J. Airey, J. Rohlf and Jr.F. Brooks, "Towards Image Realism with interactive Update Rates in Complex Virtual Building Enviroments", *Proceedings of ACM Symposium on Interactive 3D Graphics*, 1990, pp. 41-49

[55] H. Zhang et al., "Visibility culling using hierarchical occlusion maps", *Proceedings of ACM SIGGRAPH*, 1997, pp. 77-88

[56] C. Goral et al., "Modeling the interaction of light between diffuse surfaces", *Computer Graphics* Jul, 1984, vol. 18, no. 3, pp. 213-222

[57] W. Reeves, "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects", *Proceedings of ACM SIGGRAPH*, 1983, pp. 359-376

[58] J. Clark, "Hierarchical geometric models for visible surface algorithms", *Communications of the ACM*, Oct, 1976, vol. 19, pp. 547-554

[59] T. Funkhouser and C. Séquin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual enviroments", *Proceedings of ACM SIGGRAPH*, Aug, 1993, pp. 247-254

[60] D. Luebke et al., "Level of Detail for 3D Graphics", *ISBN: 1558608389, Morgan Kaufmann*, Jul, 2002

[61] S. Zelinka and M. Garland, "Permission Grids: Practical, Error-Bounded Simplification", *ACM Transactions on Graphics*, Apr, 2002, vol. 21, no. 2, pp. 1-25

[62] M. Garland, "Multiresolution Modeling: Survey & future opportunities", *Eurographics 1999, State of the Art Report*, Sep, 1999

[63] P. Lindström et al., "Real-Time, Continuous Level of Detail Rendering of Height Fields", *Proceedings of ACM SIGGRAPH*, Aug, 1996, pp. 109-118

[64] M. Duchaineau et al., "ROAMing Terrain: Real-time, Optimally Adapting Meshes", *Proceedings of the Conference on Visualization '97*, Oct, 1997, pp. 81-88

[65] E. Catmull, "A subdivision algorithm for computer display of curved surfaces", *PhD Thesis*, *University of Utah, USA*, 1974

[66] N. Greene, M. Kass and G. Miller, "Hierarchical z-buffer visibility", *Proceedings of ACM SIGGRAPH*, 1993, pp. 231-240

[67] F. Crow, "Shadow Algorithms for Computer Graphics", *Proceedings of ACM SIGGRAPH*, Jul, 1977, pp. 242-248

[68] L. Williams, "Casting curved shadows on curved surfaces", *Proceedings of ACM SIGGRAPH*, Aug, 1978, pp. 270-274

[69] T. Whitted, "An improved illumination model for shaded display", *Communications of the ACM*, Jun, 1980, vol. 23, no. 6, pp. 343-349

[70] E. Haines, "Soft planar shadows using plateaus", *Journal of Graphics Tools*, 2001, vol. 6, no. 1, pp. 19-27

[71] T. Lokovic and E. Veach, "Deep Shadow Maps", *Proceedings of ACM SIGGRAPH*, Aug, 2000, pp. 385-392

[72] S. Woodcock, "Game AI: The State of the Industry 2000-2001: It's Not Just Art, It's Engineering", *Game Developer Magazine*, Aug, 2001

[73] C. Campbell and G. McCulley, "Terrain Reasoning Challenges in the CCTT Dynamic Environment", *Proceedings of the 5th Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, 1994, pp. 55-61

[74] G. Miller, "The Definition and Rendering of Terrain Maps", *Proceedings of ACM SIGGRAPH*, Aug, 1986, pp. 39-48

[75] H. Hoppe, "Progressive meshes", *Proceedings of ACM SIGGRAPH*, Aug, 1996, pp. 99-108

[76] J. Hammes, "Modeling of ecosystems as a data source for real-time terrain rendering", *Digital Earth Moving, First International Symposium, DEM 2001, Manno, Switzerland, Proceedings*, Sep, 2001, pp. 98-111

[77] A. Lindenmayer, "Mathematical Models for Cellular Interaction in Development", *Journal of Theoretical Biology*, 1968, vol. 18, pp. 280-315

[78] C. So, G. Baciu and H. Sun, "Reconstruction of 3D Virtual Buildings from 2D Architectural Floor Plans", *ACM International Symposium on Virtual Reality Software Technology*, Nov, 1998, pp. 17-23

[79] P. Müller and Y. Parish, "Procedural Modeling of Cities", *Proceedings of ACM SIGGRAPH*, Aug, 2001, pp. 301-308

[80] M. Harris and A. Lastra, "Real-Time Cloud Rendering", *Computer Graphics Forum (Eurographics Proceedings)*, Sep, 2001, vol. 20, no. 3, pp. 76-84

[81] M. Harris et al., "Simulation of Cloud Dynamics on Graphics Hardware", *Proceedings of Graphics Hardware*, Jul, 2003, pp. 92-101

[82] J. Schpok et al., "A Real-Time Cloud Modeling, Rendering, and Animation System", *Symposium on Computer Animation*, Jul, 2003, pp. 160-166

[83] J. Rubin, "Great Game Graphics... Who Cares?", *http://www.gamasutra.com/features/20030409/rubin_01.shtml*, *Game Developers Conference*, 3rd Apr, 2003

[84] C. Cruz-Neira et al., "The CAVE: Audio Visual Experience Automatic Virtual Environment", *Communications of the ACM*, Jun, 1992, vol. 35, pp. 65-72

[85] S. Rabin (Ed.), "AI Game Programming Wisdom", *ISBN: 1584500778, Charles River Media, Inc.*, Mar, 2002

[86] R. Hayes and B. Feenstra, "Video-speed electronic paper based on electrowetting", 25th Sep, 2003, Nature, vol. 423, pp. 383-385

[87] S. Gallagher and S.H. Park, "Innovation and Competition in Standard-Based Industries: A Historical Analysis of the U.S. Home Video Game Market", *IEEE Transactions of Engineering Management*, Feb, 2002, vol. 49 no. 1, pp. 67-82

`NOTE: The author of this Thesis has archived all the referenced WWW-pages in case they get removed from the Internet. You can reach the author by email: vhelin@hut.fi / vhelin@iki.fi.`

Appendix A: The IPS in PLG

```
in project lecherous gnomes rumors are messages that creatures
pass to each other to gain information about the world they
live in. here i've tried to make a simple model of how rumors
propagate and affect the lives of the creatures who spread them.
i have no scientific basis for my reasonings and models so they
are highly heuristic and subjective. take this with a grain
of salt. all comments are welcome!


SYSTEM OVERVIEW:


the creature

every creature C_i has separate memories for rumors R that he's
heard and ready to pass on, old rumors O that C_i has heard and
acted upon (this list must be maintained, because otherwise the
creature C_i can hear (and act upon) the same rumor R_n over and over
again) and a list of experiences (rumors also) that C_i has
experienced and is also willing to tell about.

the creature also has a list N of names of other creatures,
and attached to every name there is a value n_i indicating
the respect the creature feels for that name. every name in
N can appear in the rumor messages R_i. if the creature hears
a rumor about a new person P_i then the name of P_i is added
to the creature's namelist N. every time two creatures share
rumors they also tell their names with each other, so the
creatures get to know each other quite quickly.


the rumors

every rumor consists of subject, object and action. e.g.
"orc1003 stole jasmin's gloves". the actions are
separated in the code (or lists) so that handling of each
action is handled individually (as parsing each action might
require different kind of operations). subject and object
are creatures' id numbers.

rumors are generated by hearing them from another creature,
or by witnessing a scene that generates them (e.g. creature C_a
sees creature C_b drinking lots of cola).

attached to a rumor message there is also the teller's id.
so in time some rumors might get discarded by the fact that the
teller of the rumor has now become unpopular by doing bad things
(no one wants to spread rumors told by an unpopular creature,
who once was a really good man but turned bad).

every rumor has also a priority. if the topic of the rumor is
serious then it would have higher priority than a rumor that
is of common nature.


location in a rumor

some rumors might tell about a location (e.g. "there is a barrel
```

of cheese at L". here L might be coordinates (x, y), but it could also be an identification number of an area. here's an example: the program would count the steps C_a has made in L. after exceeding some threshold t depending on L without seeing food C_a would experience that "there is no food in L" and he might tell it to other creatures, if it's valuable information.

this kind of rumors must be handled differently from other rumors as here only two things matter: what was seen and who told about it. the rumor engine does this, but one must signal the type of handling via action structure's status field (more about this later).

the respect

if creature C_a hears a rumor from creature C_b and C_b has low respect in C_a's list, then the rumor doesn't affect C_a's variables much, but if C_a respects C_b much then the effect is big.

depending on the rumors the creatures hear about each other their respect for each other also changes. C_a might respect C_b a lot, but when C_a hears from C_c, who C_a also respects a lot, that "C_b took a dump in C_a's kitchen", C_a would lower his respect for C_b.

initially every creature knows only itself and respects itself completely (same as 1.0). the initial respect creature C_a feels for another creature C_b should depend on various things, e.g. C_a meets C_b. C_b is a warrior orc: how does C_a feel about orcs in general (perhaps an average of all orcs C_a knows?)? what about C_b being a warrior?

memorizing the respect changes

every time a rumor or an experience causes a creature to change the respect he feels for another creature (i.e. when he experiences the rumor/scene generating a rumor), he must record the change (to the rumor structure). this record will be used later to neutralize the effect of this rumor if the creature finds out that the rumor was false.

e.g. creature C_a hears from C_b that C_c killed C_a's friend C_d. C_a automatically lowers his respect for C_c. next C_a sees C_d and realizes the rumor C_b told him was false. now C_a neutralizes the effect of C_b's rumor (and might lower his respect for C_b for telling a lie).

the experience

for a creature to be able to say that a rumor about him is false or true, he must maintain a list of experiences that he has lived. so every scene that can generate a rumor must also generate an experience.

the false rumours

it is also possible to spread false rumors. but if a creature

C_a hears that "C_b killed C_c" (R_a), and it didn't happen (there's no record of such an experience in C_b's experience list), and creature C_b hears R_a, he will say back that "R_a is untrue" (this is an objection rumor). if C_a believes C_b more than C_i, the creature who told C_a R_a, C_a will drop rumor R_a. C_a should then cancel the effect of R_a on his respect for C_b. C_a and C_b might start to spread the negation of R_a to clear C_b's fame.

the same thing goes for actions that prove a rumor R_b to be a lie. e.g. creature C_a hears that "there is gold in (x, y)" (R_b). he goes to (x, y) and sees that there's no gold there. now he will tell everybody the negation of R_b and creatures who hear this, and have heard the original R_b, will have to decide which rumor is the right one.

the objection rumors

all rumors are initially positive. e.g. "C_a killed C_b", "C_a ate a magical shroom" or "C_a found a treasure". only when someone finds out that a rumor R_a wasn't true he starts to spread rumor !R_a, which is an objection rumor. accepting an objection rumor !R_a means that you must neutralize the effect of R_a and replace R_a with !R_a.

the decay

depending on the priority of the rumor, the creature carrying it, might forget it in some time. low priority rumors (e.g. "C_b drank lots of cola") are forgotten faster than high priority rumors (which might be never forgotten).

the intelligence of the creature should affect the decaying speed. here's one model: creatures with high int would drop low priority rumors fast and concentrate on spreading only high priority rumors. creatures with low int would spread lots of low priority rumors.

the goodness of a rumor

if C_a hears a rumor r from C_b, it first checks if it can trust C_b. if it can, then it checks who has originally created the rumor r (C_c) C_a checks his respect for C_b and C_c, and the smaller respect will tell C_a the goodness of the rumor r (as C_b might be making it up the goodness cannot exceed C_b's goodness).

asking information

the creatures might also want to ask about things from the others. this depends completely on the AI. e.g., one might want to know about water-related rumors, if he wanted to find water.

and if creature C_a heard a rumor that C_b did something, and C_a sees C_b, C_a could go and ask C_b that if the rumor was true. this can be useful if the creatures want to confirm what they have heard, e.g., one rumor (told by C_c) tells that "C_b said that there is water at (x, y)". before travelling to (x, y), C_a might want to know if this is true, and goes to C_b and asks.

```
the action records

there must be a record describing the parameters describing every action
(as seen in the rumors) and its handling and effect on the creatures. here's
the record used in project lecherous gnomes:

struct action {
  int action;           /* action id (e.g. ACTION_KILLED, ACTION_HIT, ...) */
  int status;           /* ACTION_STATUS_IS if the action is about seeing */
  float hear;           /* the weight on hearing about the action */
  float see;  /* the weight on seeing the action */
  float subject;        /* the weight on the respect for the subject */
  float object;         /*                                   object */
  float teller;         /*                                   teller */
  float original_teller; /*                         original teller */
  float severity;       /* the severity of the action (<0 -> bad, >0 -> good) */
};

all floats are [0, 1] except the range of "severity" which is [-1, 1].

here's an example of this:

struct action actions[1] = { { ACTION_SALIVATE,
                               0.5, /* to hear a rumor about salivating is nearly */
                               1.0, /* as effective as seeing someone to do it */
                               1.0, /* the person who did it */
                               0.0, /* others aren't affected by this rumor */
                               0.0,
                               0.0,
                              -0.1  /* -0.1 < 0 so this is a bad thing to do */
                             } };

the respect update models

every time a creature hears a rumor it updates its respect for those
mentioned in the rumor (subject and object), and for those involved
in the rumor propagation process (teller and original_teller. that's only
two currently, but you could maintain a list, for each rumor, of those
creatures who have passed it on).

here C_a recieves the rumor. p is a list of respects for persons C_a
knows, e.g. p[C_a] == 1.0 and p[C_b] might be 0.1 if C_a doesn't
respect C_b much. r is the rumor C_a recieves. a is the action record
of the action r->action. if C_a heard the rumor then h=1 and s=0. if
C_a saw the action (rumor) then h=0 and s=1.


the respect update model 1 (basic)

this respect update model is basic in the sense that it cannot deal well
with rumors where the outcome of C_a hearing the rumor would depend on
C_a's respect towards the subject and object of the rumor (e.g. C_b kills
C_c. now if C_a hates C_c C_a feels good, and if C_a loves C_c C_a
feels bad). rule 1 is suitable for rumors where the outcome depends
only on the action. i'll give a better model (model 2) later, but
learning model 1 will be helpful in understanding models 2 and 3.

first we have to decide if we believe the rumor at all. we take a random
number between [0, 1] and if it smaller than p[r->teller], then we
believe the rumor, otherwise we don't let it affect us and store it
```

```
into memory for the future (in case we hear it again from the same
creature so we can drop it again).

next we need to create one value, e, which tells the effect of the rumor:

e = (a->hear*h + a->see*s)*a->severity;                       (1)

so e depends on the way C_a obtained the rumor and on the
severity of the rumor.

after this we get to update C_a's respect for the creatures involved

p[r->subject] += e*a->subject;                                (2)
p[r->object]  += e*a->object;                                 (3)
p[r->teller]  += e*a->teller;                                 (4)
p[r->original_teller] += e*a->original_teller;                (5)

here we just weight the effect e on the different creatures. we must be
careful here, because r->subject might be the same as r->original teller.


the respect update model 2 (advanced)

let's update the update model (!) to a something more general. we still
need e (1) from model 1. this time we take into account C_a's respect for
the creatures appearing in the rumor (subject and object).

p[r->subject] += e*a->subject*(p[r->object] - p[r->subject]); (6)
p[r->object]  += e*a->object*(p[r->subject] - p[r->object]);  (7)
p[r->teller]  += e*a->teller;
p[r->original_teller] += e*a->original_teller;

let's take a closer look at (6) with an example:

action = ACTION_KILLED and e = -1 (to kill is a bad thing). C_a hears
that C_b (r->subject) has killed C_c (r->object). p[C_b] = 0.1 and
p[C_c] = 1.0. now

p[C_b] += -1 * 1 * (p[C_c] - p[C_b]) = -1 * (1.0 - 0.1) = -0.9
               ^ affects the subject totally (for this example)

so C_b (the bad creature, if you ask C_a) got lots of negative respect
from C_a for killing the good creature (if you ask C_a) C_c.

next let's compute the update when p[C_b] = 0.5 and p[C_c] = 0.5:

p[C_b] += -1 * 1 * (0.5 - 0.5) = 0

two equally good creatures are fighting -> well, it doesn't matter.

next let's compute the update when p[C_b] = 0.7 and p[C_c] = 0.5:

p[C_b] += -1 * 1 * (0.5 - 0.7) = 0.2

here C_b got respect for killing C_c when C_a liked more about C_b than
C_c. this isn't quite good, because here C_a supports killing of good
creatures (e.g. p[C_b] = 1.0, p[C_c] = 0.99, and C_b kills C_c). so
we need to improve on our model even more.


the respect update model 3 (the latest)
```

```
to improve on model 2 we need to consider what are the properties
of an evil and good creature. here's a trivial way to do this:

0.0 <= p[C_n] <  0.5  ->  C_n is evil
0.5 <= p[C_n] <= 1.0  ->  C_n is good

this dichotomy is in the right direction, but it's not very useful
as some creatures are more evil than the others. but what if a bad
creature C_b kills another bad creature C_c? should C_b get positive
respect for this. i think he should:

p[r->subject] += e*a->subject*(p[r->object] - 0.5);           (8)
p[r->object]  += e*a->object*(p[r->subject] - 0.5);           (9)
p[r->teller]  += e*a->teller;
p[r->original_teller] += e*a->original_teller;

action = ACTION_KILLED and e = -1 (to kill is a bad thing). C_a hears
that C_b (r->subject) has killed C_c (r->object). p[C_b] = 0.1 and
p[C_c] = 1.0. now

p[C_b] += -1 * 1 * (1.0 - 0.5) = -0.5

so evil creature C_b got negative respect for killing a good
creature C_c. what happens when p[C_b] = 0.3 and p[C_c] = 0.4.
and another example where p[C_b] = 0.4 and p[C_c] = 0.3:

p[C_b] += -1 * 1 * (0.4 - 0.5) = 0.1
p[C_b] += -1 * 1 * (0.3 - 0.5) = 0.2

so the effect is the same: in both situations the evil creature C_b
who killed another evil creature C_c, got positive respect from C_a,
but now the amount of respect depended only on the creature's, who
got killed, goodness.

what if p[C_b] = 0.1 and p[C_c] = 0.0, and another example where
p[C_b] = 0.8 and p[C_c] = 0.0?

p[C_b] += -1 * 1 * (0.0 - 0.5) = 0.5
p[C_b] += -1 * 1 * (0.0 - 0.5) = 0.5

this example shows that regardless of the killer, the reward is
the same. but if bad creature C_b kills another bad creature C_c,
will it make C_b a good creature? i think it will, but not linearly.
let's apply sigmoid function sigm(x) to (8) and (9):

p[r->subject] += e*a->subject*(p[r->object] - 0.5)*
                 sigm((p[r->subject] - 0.5) * 10);            (10)
p[r->object]  += e*a->object*(p[r->subject] - 0.5)*
                 sigm((p[r->object] - 0.5) * 10);             (11)
p[r->teller]  += e*a->teller;
p[r->original_teller] += e*a->original_teller;

what if p[C_b] = 0.1 and p[C_c] = 0.0, and another example where
p[C_b] = 0.8 and p[C_c] = 0.0?

p[C_b] += -1 * 1 * (0.0 - 0.5) * sigm((0.1 - 0.5) * 10) =
          0.5 * sigm(-4) = 0.009
p[C_b] += -1 * 1 * (0.0 - 0.5) * sigm((0.8 - 0.5) * 10) =
          0.5 * sigm(3) = 0.476

here we can see that the worse the killer the harder it is for him
to become a better member of the society.
```

```
what if p[C_b] = 1.0 and p[C_c] = 0.1, and another example where
p[C_b] = 1.0 and p[C_c] = 0.8?

p[C_b] += -1 * 1 * (0.1 - 0.5) * sigm((1.0 - 0.5) * 10) =
          0.4 * sigm(5) = 0.397
p[C_b] += -1 * 1 * (0.8 - 0.5) * sigm((1.0 - 0.5) * 10) =
          -0.3 * sigm(5) = -0.298

here we can see that the more respected the killer the more
his actions affect his fame (think about the president killing
someone and a nameless bum killing someone).


an example

creature C_a sees water at (x, y). the occasion creates an
experience to C_a. C_a tells this experience (as a rumor) to C_b,
and C_b will learn (adds to his rumors, if he believes C_a) that
there is water at (x, y).

C_b goes on with his daily life, spreading this rumor to others. at one
point in time he gets thirsty. C_b then checks through his rumor list and
finds out that C_a, who is still in C_b's favour, said that there is
water at (x, y), and the place is the closest one of the "there is water"
rumors. so C_b travels to (x, y) and finds water. this makes C_b
forget C_a's rumor, because now C_b has experienced the same thing (and
might increase C_b's respect for C_a, because what C_a told C_b was
useful and true).

in the example, replace the word "water" with a variable and you can use
the same routines for food, water, shelter, shop, monastery, etc...

initially you can drop n creatures to your world (and more whenever you
feel like it). there they will learn about the surroundings and spread
the word. after a while you'll have creatures who know a lot about
the local places, but if you have created different kinds of
personalities, not everybody will know everything. it all depends
on the ai using the information obtained via the rumor engine...


another example

we have a group of orcs O and a group of humans H. the orcs respect each
other (so every orc is a good guy to every other orc), and the humans
respect each other, but both groups disrespect each other. H_i thinks that
every orc O_i is a crook and vice versa.

now life goes on, but one day an orc O_a kills a human H_a. this action
is seen by O_b and H_b. O_b thinks that O_a did a good thing, because
O_a killed a human H_a who O_b hated. H_b on the other hand thinks that
O_a is a bigger villain than ever before, because O_a killed H_b's good
friend H_a.

all this can be achieved just by making a rule that orcs initially
disrespect humans and vice versa. as easily you could generate "mutations"
to the group by giving some orcs the ability to love every creature
(or just humans) they see. some humans they might met with might disagree
on this. with swords and other things.


the final words
```

```
the model of rumor propagation i've described here could be used as
a part of an larger ai, to create a living and changing world. i hope
to see dynamic game engines capable to handling changing situations.
i don't hope to see any more of prerendered/hardcoded games where
you only get to run inside a tube made of indestructible material
and see static creatures with static minds.


IDEAS:


- attached to every rumor there should be a list of creatures who have
  participated in spreading it. currently only the teller and the
  alledged original teller are recorded. the length of this list should
  depend on the memory-attribute of the listener, and perhaps on the ai.


RULES:


handling a lie

C_a discovers that rumor R_i is a lie. next:

1. was R_i actually C_a's own experience (e.g. "there was an
   apple at (x, y)")?
   YES -> has C_a told R_i to someone?
          YES -> remove R_i from C_a's memory and exit
2. create a negation of R_i (!R_i).
3. if R_i was told by C_b (!= C_a) then lower C_a's respect for C_b.
4. remove R_i from C_a's memory and add !R_i there.

note: C_a might want or not to spread !R_i depending on the situation.
e.g. "there wasn't an apple at (x, y)" might be a useful rumor
if the creatures wanted to keep a list of locations they've
visited while trying to locate food (so after hearing this no-one
will go to (x, y) if they believe C_a).


computing the goodness of a rumor

C_a recieves a rumor R_a. next

1. C_a checks his respect a for the teller of R_a, and his respect
   b for the (alledged) original teller of R_a.
2. the goodness of R_a is the smallest value of {respect(a), respect(b)}.

this is based on the fact that the goodness of a rumor is as high
as the respect for the least respected creature appearing in the
rumor's path of propagation. currently only the latest and the
first tellers are recorded in the rumor structure, but nothing
forbids from recording the whole path.


handling conflicting rumors

C_a recieves a new rumor R_a that conflicts with an older rumor R_b
(or C_a's experience). next:

1. C_a computes which one he can trust more, R_a or R_b
2. if C_a trusts R_b more (the new rumor R_a doens't sound so good)
```

```
   -> exit
3. the new rumor R_a sounds better
4. if R_b is a true statement (it had an effect on C_a which must
   be neutralized)
   -> neutralize the effect of R_b on C_a
5. if R_b is an objection rumor (it had already neutralized !R_b's
   effect)
   -> experience the new rumor R_a
6. remove R_b from C_a's memory and add R_a there


telling a rumor

C_a tells C_b a rumor R_a. next

1.  C_b checks if he knows C_a. if C_b doesn't then he computes
    initial respect for C_a and adds him to the list of persons
    he knows (this operation is called chadd)
2.  C_a chadds C_b.
3.  C_b will now check his respect for C_a. if this respect is
    less than the threshold t (0.5) then C_b won't listen to C_a
    -> exit
4.  C_a selects a rumor R_a he tells to C_b.
5.  C_b now computes the goodness g of R_a.
6.  if g < 0.5 then C_b won't believe that R_a is true
    -> exit
7.  C_b randomizes a value [0, 1] and if it is greater than g
    (here we check if C_b really believes the rumor)
    -> exit
7.  C_b checks if he has heard R_a before.
    YES -> exit
8.  C_b checks if he is mentioned in the rumor.
    YES -> C_b handles R_a as a personal rumor, and exit.
9.  C_b checks if he knows !R_a or has experienced R_a or !R_a.
10. if C_b knows R_a
    -> C_b handles R_a as a duplicate rumor, and exit.
11. if C_b knows !R_a
    -> C_b handles R_a as a conflicting rumor, and exit.
12. R_a was a totally new rumor for C_b. add R_a to C_b's list
    of old (handled) rumors, and new (spreadable) rumors.
13. C_b experiences the message of R_a.


handling a personal rumor

C_a recieves a rumor R_a from C_b. C_a appears in R_a. next

1. C_a checks if he knows about R_a (i.e. has C_a experienced R_a).
   NO, and R_a is a true statement -> tell C_b that R_a is not true,
       and exit
   NO, and R_a is an objection rumor -> exit
   YES, but C_a knows !R_a -> handle the conflicting rumor R_a,
        and exit
   YES -> handle the duplicate rumor R_a


telling that a rumor is not true

C_a tells C_b that a rumor R_a C_a just heard from C_b, is not true.
next

1. C_b checks which one he respects more, C_a or the creature C_c
```

```
   C_b heard R_a from.
   C_c -> exit
2. C_b neutralizes the effect of R_a, and forgets it
3. C_b experiences !R_a, and memorizes it
```

Abbreviation	Real name
3DO	3DO Interactive Multiplayer
A78	Atari 7800
AJ	Atari Jaguar
CDI	Philips CDi
DC	Sega Dreamcast
GB	Nintendo GameBoy
GBA	Nintendo GameBoy Advance
GBC	Nintendo GameBoy Color
GP32	GamePark GP32
LYNX	Atari Lynx
N64	Nintendo 64
NES	Nintendo Entertainment System
NGC	Nintendo Gamecube
PS1	Sony Playstation
PS2	Sony Playstation 2
SGG	Sega Game Gear
SMD	Sega Megadrive
SMS	Sega Master System
SNES	Super Nintendo Entertainment System
SS	Sega Saturn
TE	NEC TurboExpress
TG16	NEC TurboGrafx-16
XBOX	Microsoft XBox

Table 6.1: Abbreviations and the corresponding meanings used in this appendix

Appendix B: Video game hardware and its popularity in the USA

This appendix examines the different video game consoles and tries to find out how much technical superiority contributes to a system's popularity. It is assumed that more powerful hardware is capable of hosting technically better games than less advanced hardware, and the better the hardware the stronger the feeling of presence might be [11]. But does the goodness of the hardware contribute more to the final score than for example, the number and quality of exclusive game titles each video game console has?

Table 6.2 lists the generations three to six of video game consoles (see table 6.1 for the abbreviations). The consoles appear in order by their popularity inside the generation they belong to, the most popular being the first. The last column, labeled as "POW", tells the machine's order number when considered the computational power of the hardware.

There are quite a few things to note when considering the different architectures, especially the design of the old consoles:

- Some NES game modules came equipped with extra chips adding e.g., more colors to the game (e.g., "Castlevania III: Dracula's Curse" (1990)).

- Some SNES game modules have integrated DSPs or CPUs enabling primitive 3D graphics (e.g., "Star Fox" (1993)).
- SMD has also an integrated 3.6MHz Z80 taking care of the sound subsystem.
- SPC700 (4.1MHz) takes care of SNES's sounds.
- SNES has hardware scaling and rotation for sprites and background.
- One could buy an add-on 32X-module to SMD, which increased the amount of onscreen colors to 32k and had two 23MHz SH2 RISC CPUs and additional graphics hardware.
- One could buy an add-on CDROM-module to AJ, SMD and TG16.
- One can select the CPU speed in SNES and TG16.
- SS has two Hitachi 28.6MHz SH2 RISC CPUs, and a 11.3MHz 68EC000 taking care of the sound subsystem.
- One could buy (although in Japan only) 64DD add-on, a 64MB disk drive to N64.
- One could buy a 4MB memory expansion module to N64.
- PS2 can play PS1 games!
- PS2 has two vector units, which means that taking the most out of PS2 will require good skills in concurrent programming, and that the program can be divided into concurrent parts suitable for the processing units.
- XBOX has an 8GB hard drive.
- XBOX and PS2 can play DVD movies!
- Network adapters are available for all 6th generation consoles.
- The column "RAM" reflects the total of CPU work RAM and video RAM.

For example, the main CPU in SNES was slower than its counterpart in SMD, but SNES had superior graphics hardware, and some SNES game modules even had add-on chips like 10.7MHz "Super FX" RISC processor. Looking at the table shows that each generation was dominated by average

Generation	Console	Introduction	Media	Bits	CPU	RAM	Colors	ROM	POW
3rd	NES	Oct. 1985	Mod.	8	1.8MHz	4kB	24	8-512kB	2
1986-1990	SMS	Jun. 1986	Mod.	8	3.3MHz	24kB	32	32-512kB	1
	A78	Jun. 1986	Mod.	8	1.8MHz	4kB	16	2-64kB	3
4th	SMD	Aug. 1989	Mod./CD	16	7.6MHz	128kB	64	0.25-4MB	2
1989-1996	SNES	Sep. 1991	Mod.	16	2.7/3.6MHz	192kB	256	0.5-6MB	1
	TG16	Sep. 1991	Mod./CD	8/16	3.6/7.2MHz	72kB	256	128-1024kB	3
	CDI	1991	CD	16	15.5MHz	1.5MB	32k	640MB	4
5th	PS1	Sep. 1995	CD	32	33.9MHz	3MB	16.8M	640MB	3
1995-2002	N64	Oct. 1996	Mod.	64	93.8MHz	4/8MB	2.1M	4-32MB	1
	SS	May 1995	CD	32	28.6MHz	3.5MB	16.8M	640MB	2
	AJ	Oct. 1993	Mod./CD	64	26.6MHz	2MB	16.8M	1-6MB	4
	3DO	Oct. 1993	CD	32	12.5MHz	3MB	16.8M	640MB	5
6th	PS2	Oct. 2000	DVD	128	294MHz	36MB	16.8M	4.7GB	3
1999-	NGC	Nov. 2001	mDVD	32	485MHz	43MB	16.8M	1.5GB	2
	XBOX	Nov. 2001	DVD	32	733MHz	64MB	16.8M	8.5GB	1
	DC	Sep. 1999	GDROM	32	200MHz	24MB	16.8M	1GB	4

Table 6.2: Video game consoles worth mentioning listed by generations, ordered by their popularity

hardware. Even if the most powerful systems had technically most advanced games that evidently was not enough to guarantee the systems' success.

Looking at the second and third generation handheld consoles, the first generation containing Nintendo's "Game & Watch"-alikes, the same pattern can be seen. "GameBoy", even being clearly inferior in every aspect except the battery life, to its competitors, still managed to outsell every one of them. There are better studies analyzing the competition in video games business [87], which offer profound explanations to the behaviour of the markets, but it is clear that other factors than the computational power inside a generation dictate the system's popularity.

Things to note about the handheld console hardware:

- LYNX has sprite scaling and distortion hardware, and an additional 16bit math co-processor, making it the most powerful 2nd generation handheld console.

- GB and GBC use a special version of Z80 (GB-Z80), which lacks Z80's shadow registers (about half of the registers) and about half of Z80's mnemonics though adds few of its own and changes some of the old. The 3.6MHz Z80 in SGG also executes some of the opcodes a few T-states faster than the 4.2MHz GB-Z80 in GB, so one can say that the Z80 in SGG is at least as powerful as the GB-Z80 in GB.

- LYNX, TE and SGG have a backlighted LCD display.

- TE is actually a handheld version of TG16!

- One can select the CPU speed in GBC, GP32 and TE.

- It was possible to get a TV tuner to SGG!

Generation	Console	Introduction	Media	Bits	CPU	RAM	Colors	ROM	Battery
2nd	GB	Aug. 1989	Mod.	8	4.2MHz	16kB	4	32kB-2MB	35h/4AA
1989-2001	GBC	Nov. 1998	Mod.	8	4.2/8.4MHz	32kB	56	32kB-8MB	13h/2AA
	SGG	1991	Mod.	8	3.6MHz	24kB	32	32-512kB	6h/6AA
	LYNX	1989	Mod.	8/16	4MHz	64KB	16	128kb-2MB	4h/6AA
	TE	1990	Mod.	8/16	3.6/7.2MHz	72kB	256	128-1024kB	3h/6AA
3rd	GBA	Jul. 2001	Mod.	32	16.7MHz	384kB	32k	4-32MB	15h/2AA
2001-	GP32	Feb. 2002	SMC	32	22-133MHz	8MB	64k	-	12h/2AA

Table 6.3: Handheld game consoles worth mentioning listed by generations, ordered by their popularity

- Using a converter one can play SMS games on SGG!
- GP32 has 320x240 pixels on a 3.5" screen. GP32 uses smart media cards (SMCs) and connects to a PC via universal serial bus (USB).
- There are emulators for GP32 emulating NES, SNES, SMS, SMD, SGG, GB, GBC and many others.