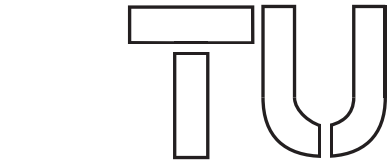


TECHNISCHE UNIVERSITÄT WIEN

Dissertation

Priority Scheduling for Networked Virtual Environments and Online Games

ausgeführt
zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Leitung von
Univ. Prof. Dipl.-Ing. Dr. techn. Werner Purgathofer,
Institut 186 für Computergraphik und Algorithmen,
und unter Mitwirkung von
Univ.-Ass. Dipl.-Ing. Dr. techn. Dieter Schmalstieg

eingereicht
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik,

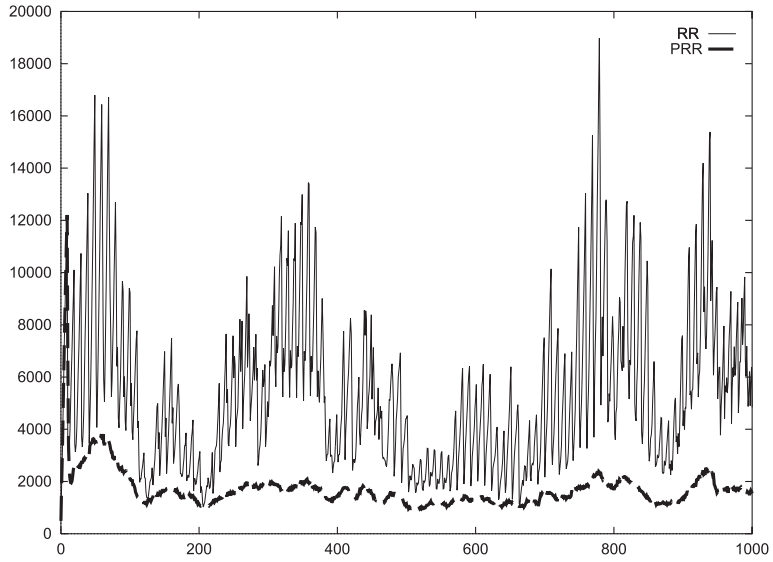
von
Dipl.-Ing. Christian Faisstnauer,
Matrikelnummer 9056328,
Horazstrasse 4/H,
I-39100 Bozen, Italien,
geboren am 12. August 1971 in Bozen, Italien.

Wien, am 18. Mai 2001

Chris Faisstnauer

Priority Scheduling for Networked Virtual Environments and Online Games

(PhD Thesis)



http://www.cg.tuwien.ac.at/faisst/
mailto:faisst@cg.tuwien.ac.at

Abstract

The problem of resource bottlenecks is encountered in almost any distributed virtual environment or networked game. Whenever the demand for resources – such as network bandwidth, the graphics pipeline, or processing power – exceeds their availability, the resulting competition for the resources leads to a degradation of the system's performance.

In a typical client-server setup, for example, where the virtual world is managed by a server and replicated by connected clients which visualize the scene, the server must repeatedly transmit update messages to the clients. The computational power needed to select the messages to transmit to each client, or the network bandwidth limitations often allow only a subset of the update messages to be transmitted to the clients; this leads to a performance degradation and an accumulation of errors, e.g. a visual error based on the positional displacement of the moving objects.

This thesis presents a scheduling algorithm, developed to manage the objects competing for system resources, that is able to achieve a graceful degradation of the system's performance, while retaining an output sensitive behavior and being immune to starvation. This algorithm, called Priority Round-Robin (PRR) scheduling, enforces priorities based on a freely definable error metric, trying to minimize the overall error. The output sensitivity is a crucial requirement for the construction of scalable systems, and the freely definable error metric makes it suitable to be employed whenever objects compete for system resources, in client-server and peer-to-peer architectures as well. Therefore Priority Round-Robin scheduling is a substantial contribution to the development of distributed virtual environments and networked online-games.

Kurzfassung

Ein Mangel an Resourcen ist ein Problem, das bei nahezu allen Virtual-Reality Applikationen (’Virtual Environments’) und Videospielen beobachtet werden kann. Die Netzwerk-Bandbreite, der Durchsatz der Graphik-Pipeline, oder die verfügbaren Prozessorzyklen reichen oft nicht aus, um die Anforderungen des Systems zu erfüllen. Der daraus folgende Wettstreit um die Resourcen führt zu einer massiven Beeinträchtigung des gesamten Systems, und beschränkt dessen Skalierbarkeit.

In einer Client-Server Anwendung, zum Beispiel, wird die ’virtuelle Welt’ von einem zentralen Server verwaltet, und von mit dem Server in Verbindung stehenden Clients repliziert, welche außerdem eine graphische Darstellung für den Benutzer erzeugen. Dies erfordert, daß alle Clients vom Server über sämtliche Änderungen in der gemeinsamen Datenbank informiert werden. Die zum Erzeugen dieser Mitteilungen benötigten Prozessorzyklen, oder die zum Übermitteln benötigte Netzwerk-Bandbreite, übersteigt oft die zur Verfügung stehenden Resourcen des Systems, sodaß nur eine Teilmenge der erforderlichen Daten bearbeitet werden kann. Dies führt zu einer Aufsummierung von Fehlern, z.B. Sichtbarkeitsfehler, welche auf Positionsänderungen der sich bewegenden Objekte beruhen.

Diese Dissertation präsentiert einen Scheduling-Algorithmus, genannt Priority Round-Robin (PRR) Scheduling, welcher zur Verwaltung von um Systemresourcen konkurrierende Objekte entwickelt wurde. PRR weist sämtlichen Objekten eine Priorität zu, welche anhand einer frei definierbaren Fehlermetrik bestimmt wird, und versucht den Gesamtfehler des Systems zu minimieren. Trotzdem ist der Aufwand des Algorithmus nur von der Anzahl der auszuwählenden Objekte abhängig, und nicht von der Gesamtanzahl der Objekte (’output sensitive’). Außerdem garantiert der Algorithmus, daß alle Elemente mindestens einmal innerhalb einer gewissen Zeitspanne ausgewählt werden, was die Gefahr einer ’Starvation’ minimiert.

Durch die frei definierbare Fehlermetrik kann PRR in nahezu allen Situationen eingesetzt werden, in denen es zu Engpässen bei der Zuteilung von Resourcen kommt, und durch die ’output sensitivity’ des Algorithmus wird die Konstruktion von skalierbaren Systemen wesentlich erleichtert. PRR ist in der Lage, den durch Resource-Engpässe erzeugten Fehler stufenlos zu minimieren, und passt sich laufend an dynamische Situationen an.

Acknowledgements

While it will be impossible to thank all those who have contributed in some way to this work, there are certain folks who must be acknowledged. First and foremost are the members of the Institute of Computer Graphics and Algorithms at Vienna University of Technology, where this work was undertaken, especially my advisors Dieter Schmalstieg and Werner Purgathofer. Thanks to Werner also for giving me a chance to work at the institute, and to Meister Eduard Gröller for spiritual support throughout the whole project.

I would like to thank my family and friends who have encouraged and supported me over the many years spent on this and previous work. My family has never wavered in their support, or in their conviction that I could and would one day complete this dissertation.

This work was supported by the European Community under contract no. FMRX-CT-96-0036.

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

J.R.R. Tolkien (1892-1973)
English poet and novelist.

Contents

Chapter 1

Introduction

Sitting inside the cockpit of his new and carefully tuned gladiator robot, the japanese mercenary Tetsuo scans the surroundings for approaching enemies. Suddenly the radar starts to emit warning sounds, and Tetsuo spins the giant robot around to face the incoming thread. What he sees does not leave him much time to react, because the other BattleMech heads toward him at incredible speed. Suddenly something strange seems to happen to the unknown opponent, as he starts to perform a staccato-like motion, appearing to stand still for a few fractions of a second, for then abruptly reappearing a few meters ahead of its previous position. Confused by this strange phenomenon, Tetsuo nearly missed to see the missile that suddenly materializes halfway between him and his opponent, aiming directly at the torso of his BattleMech. Now where did this missile come from? Did the other Mech shoot it? Tetsuo directs his robot into a frenetic jump into the air, to avoid the missile, and fortunately manages to skip it by inches. As this calls for an appropriate response, Tetsuo in turn triggers two small but fast projectiles towards his opponent. And again, the other BattleMech seems to be protected by an invisible force. As he appears to be completely immobile, the first projectile passes right through him, without doing any damage, and microseconds later he makes again an invisible motion some meters to the side. But the second projectile hits its target, and apparently in a vulnerable part, as a series of small explosions starts to cover the other BattleMech, transforming it into a cluster of debris. This was the time Tetsuo relaxed too early, removing his hands from the commands, as unexpectedly another enemy missile appeared in mid-air, and like a shot from afterlife convicts Tetsuo's BattleMech into a clearly disassembled state.

This might be an excerpt from a networked online game, in which the user has to maneuver a humanoid robot through a virtual scenery, to compete with other users in a gladiator-like competition. In such a distributed virtual environment, geographically distant users have the illusion to be all together at the same time in the same place - a virtual arena where to engage in challenging tournaments. This scenario resembles a popular online-game called 'BattleTech', which was one of the first highly successful networked games set up in entertainment centers. The screenshot on the previous page is from 'MechWarrior 4', a PC-based conversion of the BattleTech concept.

However, this example also describes some of the effects that might be caused by a lack of resources, e.g. if the demand for network, rendering or processing resources exceeds their availability. The first anomaly described above, the 'staccato-like motion', is a typical effect that derives from a delayed or missing transmission of update packets. As each host relies on such messages to be informed about the actions of the remote entities managed by other hosts, an unsteady flow of information substantially affects the representation of these entities and the interaction with them. This is also valid for the missile 'materializing halfway between both opponents': if the message describing the firing of the missile by the remote entity does not arrive in time (or does not arrive at all), the first information the local host receives about this action is a missile coordinate in midair. The host cannot interpolate the missing motion of the missile (to display the user a delayed representation), or ignore it at all, because the missile is an 'influential' object whose knowledge is essential to the player; hence every information about it must be displayed immediately after reception.

An unsteady or delayed flow of update messages highly affects interaction between the objects in the environment; the user has a delayed perception of the actions triggered by the remote users (such as the incoming missile). It also prevents the user from correctly targeting the other players, because their position as assumed by the local host is outdated, and the projectiles will be aimed at an afterimage. The 'shoot from afterlife' described above is a phenomenon that was reported from several distributed environments, such as NPSNET ([httpNPS]) and various online games. If the opponent shoots a missile shortly before being destroyed, and the delivery of the message is delayed enough for the remote users to be informed about the destruction of the opponent before receiving that message, it appears like the missile was fired from an inoperative entity.

The illusion of immersion, and the illusion of being all together at the same time in the same place largely depends on the performance of the system: fast interaction rates, smooth animation and fluid sound, for example, are essential components, and a lack of those resources is likely to foil the realism of the virtual world.

The networked online game described above is just one of the many possible areas of application for distributed virtual environments, which can provide much more intuitive interaction metaphors than typical desktop setups, e.g. by allowing the use of innate abilities like communication via gestures. Unfortunately, the

resource demands for virtual environments are very high, often leading to scalability problems and undesirable performance decreases due to a lack of available resources.

The following work focuses on dealing with these resource bottlenecks. First the structure of networked environments is examined, to highlight the bottlenecks that might be encountered, for then introducing a resource management technique - a priority-based scheduling algorithm - that is able to optimize the usage of the resources and achieve a graceful degradation of the system's performance.

1.1 Task of a distributed virtual environment

A distributed virtual environment (DVE), often also referred to as networked virtual environment (NVE), is typically represented as a set of independent entities, each having an own geometry and behavior. Some entities are static (such as buildings or terrain), other have dynamic behavior that is either autonomous (e.g., drones) or controlled by a user via input devices (e.g., vehicles or the gladiator robots mentioned in the introductory game). Every entity is managed by a host, either automatically via the system's software, or controlled by the user via the aforementioned input devices which can range from mouse, keyboard or joystick to data gloves or motion trackers. Output devices connected to the host allow the user to perceive the environment, typically from the point of view of the controlled entity which is the user's representation in the environment. Frequently used output devices are e.g. monitors, head-mounted displays, headphones, force-feedback devices or motion platforms.

Distributed virtual environments, as compared to non-networked environments, have the task to bridge the geographic distance between the users. The hosts are connected to a network over which they can exchange messages; this allows remote users to participate and cooperate in a common environment and perform what is called a 'distributed interaction' or 'distributed simulation'.

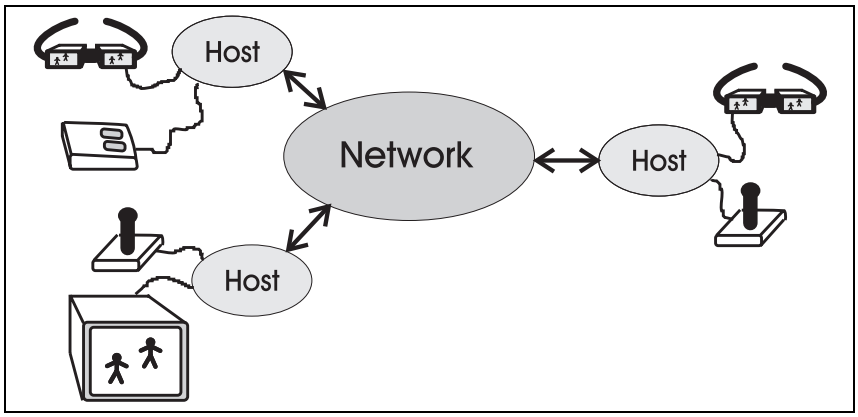


Figure 1.1: Basic structure of a distributed virtual environment.

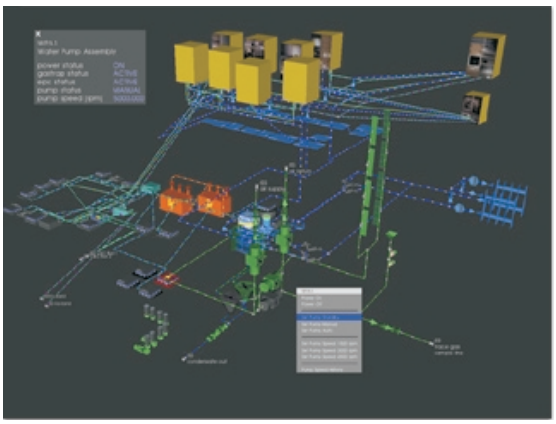
The system should allow the users to interact with each other and the environment in an intuitive and realistic manner; they should have the impression of being at the same time in the same place. This feeling of 'presence' depends on the input and output devices available, the interaction and navigation metaphors selected, and the performance of the system, which includes factors such as frame rate and response time. Basically, the more intuitive is the input and the more realistic the output, the greater is the feeling of immersion. The scope of the environment determines the desired level of realism, but in most cases the environment should provide a comprehensive illusion of immersion.

1.2 Examples of virtual environments

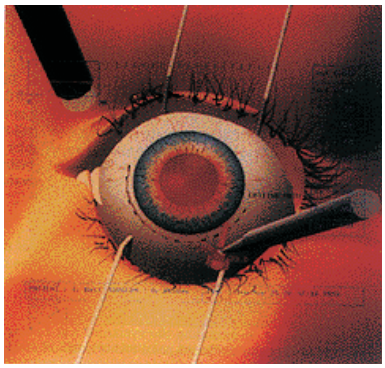
A virtual environment (VE) arguably provides the most natural means of communicating with a computer because it allows us to use our inherent 3D spatial skills that have evolved over thousands of years (such as walking, gesturing, looking, grabbing etc.) There is a vast range of potential applications for virtual environments, for example safely *training* personnel for high-risk activities (e.g. astronauts or pilots), or supporting rehearsal possibilities for *medical* personnel like virtual surgery (see Figure 1.2(b)). VR can be employed in the context of *information visualization*, to provide a comprehensive overview over the data at disposition (Figure 1.2(a) shows a diagnostic system for the International Space Station). It can be helpful to understand complex phenomenons, such as in the NASA Ames' Virtual Windtunnel shown in Figure 1.2(c), which allows to visualize and manipulate computational fluid dynamics models. Using VE technology, *architects* is given the possibility to design and walk through virtual buildings, rooms or cities (Figure 1.2(d)); it might also be a new medium for *artists* to express their visions.

Many of the applications in the areas mentioned above are not implemented in a distributed fashion, because the tasks of building a virtual house or experimenting with the Virtual Windtunnel do not necessarily need network support. There are however research areas in the context of virtual reality that benefit extensively from the ability to bridge the distance between the participating users.

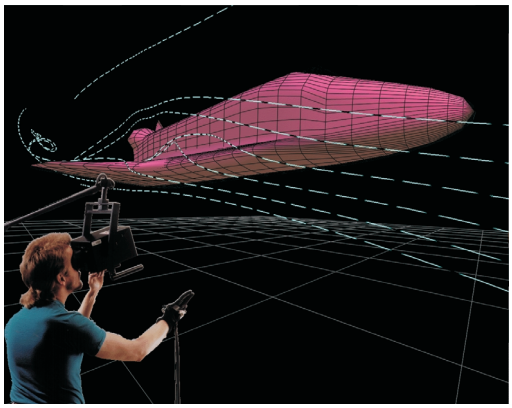
For example in *collaborative research*, where geographically distant engineers can operate on the same project by constructing, examining and manipulating shared virtual objects in a common environment. Or *collaborative business*, where teleconferencing and 'virtual' meetings (Figure 1.3(b)) allow businessmen to avoid time-consuming travels. 'Computer Supported Cooperative Work' (CSCW) is a term often used in this context. The interaction metaphors introduced by VE systems, combined with the ability of networks to bridge large geographic distances, allow to explore and manipulate environments which are remote or hazardous to human beings, for example nuclear or chemical power plants, quarantine zones, caves, volcanos, the bottom of the sea or even distant planets. Figure 1.3(a) shows an example of such a *teleoperation* device, that translates the motions of a human



(a) Diagnostic system for the ISS. © Imagination computer services G.m.b.h, Austria.



(b) Virtual eye surgery.



(c) Virtual Windtunnel. © NASA Ames research center.

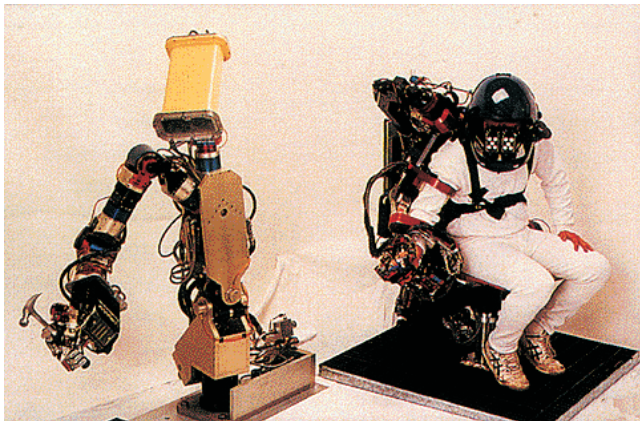
(d) Architectural walkthrough.

Figure 1.2: Examples of Virtual Environments.

operator to a remote place, and vice versa lets the user perceive the environment as if he were there personally.

A similar area of application is *remote surgery*, where a robot is telecontrolled by a surgeon to perform an operation on a distant patient; this might be especially useful if one medic has to care about many patients in different locations contemporaneously, or if it is e.g. time critical to rely on the abilities of a specialized surgeon residing in a remote location.

Extensive efforts have been put in the development of distributed virtual environments to perform large scale military *training* of soldiers; the goal is to allow a large number of geographically distant users to contemporaneously participate in virtual exercises and combat maneuvers. This permits to simulate maneuvers of an extent not possible in the real world, and allows to model any desired terrain and environment, other than including experimental vehicles or airplanes only existing



(a) Teleoperation device following the user's movements.

(b) Meeting in a virtual room.

(c) Military simulation. © LORAL industries.

Figure 1.3: Examples of Virtual Environments.

on the drawing board. Furthermore, the stress on the environment and the peril of injuries is almost nonexistent.

And last, but not least, distributed virtual environments are being employed for *entertainment* purposes at a rapidly growing rate. The manifold possibilities of modelling a virtual surrounding, and the ability of multiple players to interact in the same game, broods interest in children and adults similarly. Figure 1.4(a) shows the 'Loch Ness' adventure, in which crewmembers have to collaborate to maneuver a submarine through the lake of Loch Ness, in order to save Nessie's eggs from hunters. 'BattleTech' is a game developed mainly for adults, where different players engage in tournaments by maneuvering gladiator robots through a common environment (such as the figure on the first page). Figure 1.4(b) depicts a cluster of simulators used to control these so called 'BattleMechs'.

There also exist various multiplayer PC games, played over the internet, that can to some extent be defined as distributed virtual environments, although - mainly due to hardware restrictions - the interaction rates and the feeling of immersion

(a) The Loch Ness adventure. © Iwerks Entertainment / Evans & Sutherland.

(b) BattleTech cockpit. © Virtual World Entertainment.

Figure 1.4: Examples of Virtual Environments

are usually quite limited. Some popular categories are first-person shooters, strategy games and role-playing games. First-person shooters, such as Doom, Quake ([httpID]) or Half-Life ([httpHALF]), are based on fast infights in a rather small scenery, while the task in strategy games usually consists in planning and directing a battle in a large environment, using various types of soldiers, vehicles and weapons; the interaction is sometimes interrupted to plan the next moves. Role-playing games contain action as well as strategy components; the goal mostly consists in exploring a large environment to solve determined quests. Commercial online role-playing games, such as Ultima Online ([httpUO]) or Everquest ([httpEVER]), are based on environments resembling whole worlds, and are often populated with thousands of avatars; therefore they show all bottleneck problems of a typical distributed virtual environment.

1.3 Challenges of distributed virtual environments

The requirements for distributed virtual environments are manifold.

From the *user's point of view*, they (typically) should provide a comprehensive sense of *realism* and *immersion*. The illusion of immersion depends on many factors: first of all, the user must be able to interact intuitively with the environment, which largely depends on the interaction metaphors and input devices employed. For example, selecting an object by pointing at it with a glove is certainly more intuitive than selecting it from a drop-down menu using a mouse, especially if the whole operation is performed in a 3D environment. Concerning the output devices

available, a HMD equipped with headphones providing spatial sound leads to a much better feeling of immersion than sitting in front of monitor with monotonic loudspeakers.

But this describes only the 'front-end' of the environment. The system itself must allow truly real-time, *interactive* manipulation of the environment; changes to the environment must be perceived immediately. The animations are required to be smooth, and realistic sound is highly desirable. All users should be able to interact as if they were all just centimeters away from the tip of each other's nose, independent from the physical distance to the other participants.

From the *system's point of view*, some of the most important requirements are *scalability*, *fault tolerance*, and *authentification* or *accounting* services. The fault tolerance of a system determines its response in case of malfunction of one or more components. In the worst case, the whole system blocks or crashes. An effective fault tolerance strategy should limit the effect to the faulty components, bridging them so that the users do perceive few or no performance decreases. Authentification and accounting services are an area that is especially important in commercial distributed environments, where security concerns and a correct billing is essential to user satisfaction. The scalability of a system is maybe the hardest point to achieve. Distributed virtual environments put heavy requirements on resources, and each additional user or new object increases the load. To achieve scalability, it is absolutely necessary to optimize the usage of the available resources.

Providing an efficient resource management is a crucial requirement in constructing distributed virtual environments. The purpose is not only to minimize the effects of the inevitable *resource bottlenecks* that arise whenever the demand for the resources exceeds their availability, but it also helps making the system more fault tolerant, and increases its scalability. The less demand each object in the environment puts on the system, the more users can be accommodated, and the less influence has the failure of a component.

Some of the main bottlenecks, which degrade the system's performance, are:

- The **network**, which handles the transmission of messages and the exchange of information between the participating hosts.
- The **graphics system**, responsible for rendering the images.
- The **processing cycles** required to perform the computations necessary to run the environment (e.g. simulating objects or processing the users' input).

Compared to a non-networked virtual environment, a distributed VE has the advantage of being able to bridge geographic distances, but it also introduces the notion of 'distribution' into all aspects of the system. Hence the developers have to deal with the issue of:

- A **distributed database**: other than requiring a generic knowledge concerning the implementation of databases (e.g., efficient storage and data query), the notion of distribution requires to additionally manage the aspects of replication and consistency among the various parts of the database.

- As a realistic interaction between the user and the system requires an immediate response to every user's action, a **distributed interaction** has to deal with the delay introduced by the network, in addition to the requirements for intuitive interaction metaphors.

- Performing a **distributed simulation** or **computation**, in order to partition the processing load on all resources available, involves areas such as parallelization and process synchronization.

- The **authentification** of users or objects in a distributed environments also involves distributed database access and synchronization.

A distributed virtual environment relies on the network to exchange information (by sending and receiving messages), but the usage of a network always introduces a delay - as small is it might be; the information exchange can never be simultaneous. It is one of the tasks of the system to mask the artefacts due to its distributed nature. Therefore an efficient resource management is especially crucial if the environment is distributed - with special attention paid to the network itself.

All existing DVE employ various techniques to avoid and manage resource bottlenecks. We distinguish between two basic approaches:

- **Reduction techniques**: their task is to reduce the 'load' on the resource, by reducing the number of objects[1] competing for it. These approaches involve compression, Area-of-Interest management (e.g. visibility culling), or exploiting user perceptual limitations such as Levels of Detail (LOD). The terms '**reduction techniques**' and '**filtering techniques**' are often used synonymously, and for the sake of simplicity we will continue this tradition. However, we must be aware that in a strict sense the term '*filtering technique*' is a correct denomination only for Area-of-Interest approaches, and to some extent also for Dead Reckoning algorithms; techniques such as compression or LOD do not apply filtering functions on the competing elements. Chapter 4 provides a description of the various reduction techniques; additional starting points can be found in [Clar76] (culling), [Gree97] (a generic Area-of-Interest model) and [Heck97] (geometric Levels of Detail).

[1] Please note that in this context the term 'object' is intended as an individual element competing for a resource; it must not necessarily coincide with a geometric object in a virtual environment that can generate several elements demanding a resource. However, for not extending the terminology excessively, we will limit ourselves to the term 'object'.

- Employing **scheduling techniques** to repeatedly select which of the competing objects should be granted the resource requested; the choice among the competing objects is based on one or more determined system parameters.

Both approaches have their limitations: if reducing the number of competing objects (e.g. removing the objects which are invisible from the rendering pipeline) still generates more objects than the resource can manage, the bottleneck problem persists. In turn, a scheduling technique is always only an *approximation* of the optimum state, as only a subset of the competing objects can be chosen; some objects must always be neglected. Hence the fewer objects are competing, the better. The best results can be achieved by combining reduction and scheduling techniques.

1.4 Contribution

We contribute a scheduling algorithm to the set of available resource management techniques that is not only able to enforce priorities while being at the same time output sensitive, but that also allows to efficiently combine scheduling and reduction/filtering approaches. This technique - called **Priority Round-Robin (PRR)** scheduling - can decrease the load on the resources and provide a graceful degradation of the system's performance, and due to its output sensitivity ease considerably the construction of truly scalable virtual environments. PRR is starvation free and enforces priorities based on a freely definable error metric (the algorithm tries to minimize the overall error); hence it can be employed whenever objects are competing for a resource. By including reduction techniques (e.g. visibility culling) in the determination of the objects' priorities, PRR is able to fill the gap between reduction and scheduling techniques, and at the same time preserve an output sensitive behavior.

A popular approach to build virtual environments (and especially online games) is to use a client-server architecture: the virtual world is managed by the server and replicated by connected clients, which visualize the scene and/or navigate an avatar through the environment. All updates from the client are routed via the server (often also responsible for the simulation of autonomous entities), which can perform arbitrary reduction/filtering functions. Some systems employ visibility information in order to decrease the network load, by transmitting each client only updates for those objects visible to it. Timely delivery of update messages is essential to avoid visual errors (e.g., server and client having a different position information for the same object).

However, these approaches cause a substantial overhead to the server, as it is often required to examine all objects in the environment for each client. For example, to transmit only the visible object updates to a client, it is necessary for the server to keep track of the point of view for all clients, and continuously select the corresponding visible objects. Assume $N = number\,of\,clients = number\,of\,objects$.

Examining all objects for every client leads to an effort of $O(N^2)$, which substantially affects the scalability of the system. Furthermore, these reduction techniques do not deal with the issue of scheduling the remaining objects. If the number of messages to be transmitted still exceeds the network bandwidth, the bottleneck problem persists. In this case (or if no reduction/filtering is employed at all), we face a scheduling problem similar to those found in operating systems research.

But scheduling in operating systems is not identical to scheduling in VEs. In particular, VEs can host a very large number of objects, so that examination of every object in every turn is too computationally expensive. Instead, applications in a VE requires an *output sensitive* algorithm that operates with constant effort per connected client, hence being dependent only on the number of objects to schedule, rather than the number of objects in the environment (which would be *input sensitive*). The simple Round-Robin (RR) approach to scheduling has this property and is therefore often used for such scheduling problems. But the RR strategy - simply selecting every object in turn - cannot accommodate dynamically changing simulations. For example, if the server has to distribute updates of entities moving with variable speed, for increased realism in the behavior, fast entities will require more frequent updates than slower moving ones. Such priorities cannot be achieved with plain RR.

The PRR algorithm explained in detail in Chapters 6 ff. is able to enforce priorities, while retaining the output sensitivity and starvation-free performance of RR; hence PRR is a valid replacement for RR in most circumstances. We will evaluate the algorithm in the aforementioned client-server system and compare its performance to plain Round-Robin. PRR is used to schedule the update messages at a constant effort of $O(k)$ per client, where k is the number of updates that can be transmitted by the network (and thus have to be selected); the priority of the objects is determined by the visual error, e.g. the position displacement. By applying visibility culling when the objects are selected by PRR (and including the visibility information in the determination of the objects' priorities), the resulting effort is still $O(k)$. Hence we have an overall effort of $O(k * N) = O(N)$ for N connected clients, an output sensitive behavior which is crucial for scalable environments. The performance increase achieved by PRR will be determined from comparing the visual error of PRR to the visual error of RR, when both are used to schedule the update messages that the server transmits to the clients. Although we evaluate the PRR algorithm in a client-server system (for reasons discussed in more detail in SubSection 2.3.2 and Chapter 9), PRR scheduling is also applicable to peer-to-peer systems, with the known difficulties of employing filtering techniques in serverless systems (refer e.g. to Section 4.1)

Publications

In the last couple of years we have already published several papers related to the PRR algorithm. This thesis provides a comprehensive conclusion of this work, covering the theoretical aspects of the algorithm, and including an extended evaluation section. The publications in question are:

- [Fais00a] C.Faisstnauer, D.Schmalstieg, W.Purgathofer: Priority Round-Robin Scheduling for Very Large Virtual Environments. *Proceedings of the VR'2000 conference*, pp. 135-142, 2000.
  This paper was awarded the honorable mention in the Best Paper Award, and also selected for republication in the *Virtual Reality* journal published by Springer.

- [Fais00c] C.Faisstnauer, D.Schmalstieg, W.Purgathofer: Scheduling for Very Large Virtual Environments and Networked Games Using Visibility and Priorities. *Proceedings of the DIS-RT 2000 conference*, pp. 31-38, 2000.
  An extended version of this paper will be republished in one of the future issues of the *SCS Transactions* journal.

- [Fais00b] C.Faisstnauer, D.Schmalstieg, W.Purgathofer: Priority Scheduling for Networked Virtual Environments. *IEEE Computer Graphics and Applications (CG&A)*, Vol. 20, No. 6, 2000.

The remainder of the thesis is structured as follows:

In Chapter 2 we will present an overview about the structure of distributed virtual environments, followed by a treatment of the bottlenecks encountered (Chapter 3) and the resource management techniques typically employed to deal with them (Chapter 4). Chapter 5 lists some academic and military DVE along with references to the resource management techniques applied by them.

The Priority Round-Robin (PRR) algorithm is introduced in Chapter 6, describing its theoretical background. PRR can be applied in any situation where objects compete for a determined system resource; however in our opinion network bottlenecks are the biggest limitations to the scalability of distributed virtual environments. Hence we will evaluate the PRR scheduling in a testbed resembling a large scale DVE, where it is employed to schedule the transmission of update messages over the network. As visibility information is available in most virtual environments, the ability of PRR to be combined with reduction/filtering techniques will be demonstrated by including visibility information in the determination of the object's priorities (Chapter 7). Chapter 8 describes a heuristic that allows PRR to deal with unpredictable object behavior, as it is often caused by user-controlled avatars, especially in online games.

The evaluation section consists of two parts. The testbed employed to evaluate PRR is described in Chapter 9, and the results are presented in Chapter 10.

As the evaluation section cannot possibly cover all imaginable configurations of DVE's, the research on this area will be continued. Any information about future developments can be obtained directly from the author.

Chapter 2

Main components of a distributed virtual environment

From a very general point of view, a distributed virtual environment (DVE) can be seen as a set of hosts connected through a network, and operating on a shared database that allows the objects in the environment to interact with each other. The following three subsections provide a more detailed analysis of this basic classification.

2.1 Hosts

A simple sketch of a distributed virtual environment consists of geographically distant hosts connected through a network, such as depicted in Figure 1.1, each host allowing a user to participate in the environment: by manipulating input devices such as a keyboard, joystick, or data glove, the user can maneuver an avatar through the environment; a monitor or head-mounted display (HMD) connected to the host provides the user a representation of the environment. Of course a virtual environment might consist of many different host types, some of them providing services to the environment (instead of managing users) or managing more than one user; in this first sketch the task of a host is limited to manage one determined user. Hence each host basically performs a repeated traversal of the following main loop:

1. **Read the input devices**. The user can issue commands to the system by manipulating input devices connected to the host, e.g. to control an avatar through the environment. The motion of the avatar changes the local state of the environment (as stored by the host).

2. **Read from the network**. Whenever the user manipulates its avatar, this local change to the environment is distributed over the network to the other hosts (see step 4). Thus to get information about the other avatars in the environment, each host must read incoming packets from the network and update the locally stored state of the environment accordingly.

3. **Computational modelling**. From the motion information gathered about the own (from the input devices) and the other avatars (from the network), the host can now process interactions among the objects in the environment; this may include physical modelling of local objects, predictive modelling of remote objects, or collision detection and response.

4. **Write to the network**. Each host must communicate the changes made by its local objects (such as the 'own' avatar) on the common environment to the other hosts. This is done by transmitting messages via the network, so that the other hosts can update their internal representations of the environment accordingly.

5. **Rendering of the environment**. As last step in the main loop the environment must be presented to the user. It mainly involves the generation of images, displayed via a head-mounted display or a traditional monitor, which makes up for the lion's share of the resources required in this step. But to achieve a comprehensive sense of immersion, the other senses of the user should be stimulated as well, e.g. by generating spatial sound, force-feedback, haptic information, or manipulating the sense of balance.

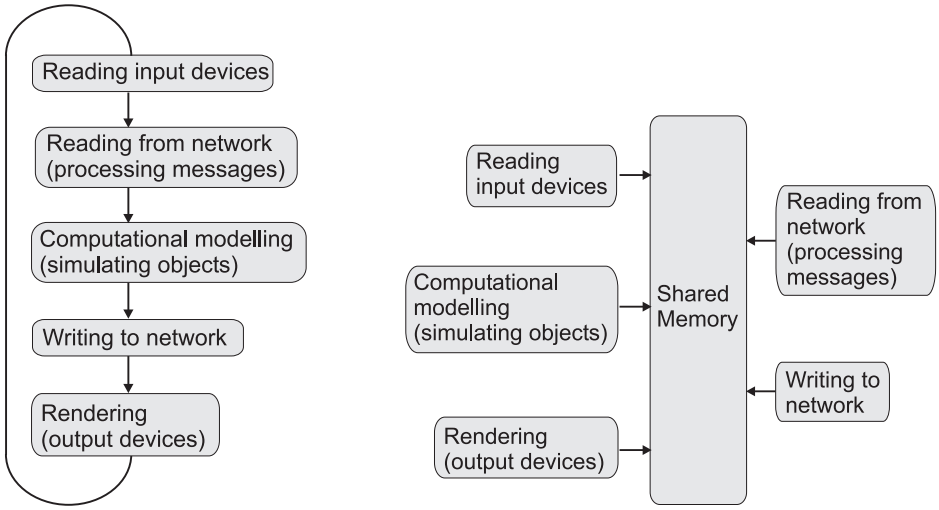
This mainloop can be implemented by the client in either a *single-threaded*, or a *multi-threaded* approach. When only one single thread is used (see Figure 2.1(a), all steps are executed sequentially, one after another. This is easier to implement than the multi-threaded version, but enables each module to slow down or block the whole system; the entire main loop is only as fast as the slowest component.

Using a dedicated thread for each module of the main loop, such as depicted in Figure 2.1(b), requires additional synchronization overhead, as the modules exchange their information over a shared memory (for more related information see e.g. [El-R98]). But it allows all modules to operate at their own speed, and supports parallel computing as well as more advanced resource sharing.

2.2 Shared database

In a distributed virtual world, users should have the illusion of being at the same time in the same place. As all hosts participate a common environment, they operate on a shared database describing the virtual scene. We mainly differentiate between two approaches of storing and managing the database, which are closely related to network structure employed: using a *central repository* vs. managing the environment in a *distributed database*.

- **Central repository**: the environment is usually stored by one or more (centralized) server(s), and replicated fully or partially by the connected clients. The server(s) are responsible for managing the database and the transmission of data to the clients. Typically the object geometry is distributed to



(a) Single-threaded main loop. (b) Multi-threaded main loop.

Figure 2.1: Single-threaded versus multi-threaded implementation of the main loop.

the clients at startup, so that communication at runtime can be limited to the transmission of update messages, hence saving network bandwidth. However, having each client store the entire environment implies heavy memory requirements; thus it is necessary to make a tradeoff between a complete download at startup, or uploading/downloading part of the object geometry at runtime. This saves memory, but puts additional load on the network.

- **Distributed database**: an opposite approach is to assign the task of managing the database and distributing the changes to the hosts themselves. Each hosts is responsible for transmitting the geometry and the state changes of the objects it manages to the other hosts, without the help of a centralized server. Thus it is necessary that the hosts negotiate the consistency of their databases among them. This approach makes it much more difficult for a host to store only a determined part of the environment, hence usually all hosts store the whole environment in their local database.

Some publications related to this topic (e.g. [Sing99]) employ the term '*central repository*' only in conjunction with a completely consistent replication of the database: all changes to the environment are perceived fully and simultaneously by all hosts. The fact that some databases might be temporarily inconsistent and only updated after a determined delay is classified as '*distributed database*'. We think that is interpretation is somehow misleading, and prefer to relate this classification to the physical architecture of the database, by talking of '*central repository*' in conjunction with client-server networks, while the term '*distributed database*' is related to peer-to-peer networks.

2.3 Network

2.3.1 Architecture and Protocols

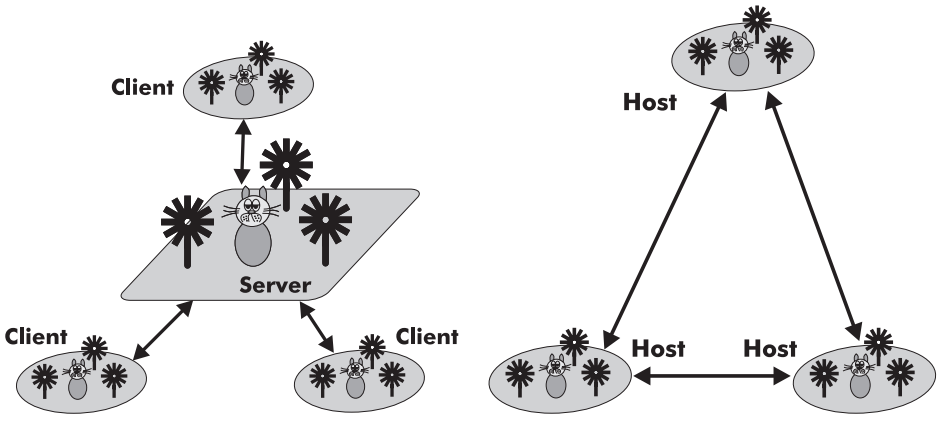
The network is the core component of a distributed virtual environment: the communication between the hosts can only take place using a network to exchange messages. As the use of a network always introduces a delay (a truly simultaneous transmission is not possible), it is important to keep the number of messages to be transmitted as low as possible.

The **network structure** employed can heavily affect the performance of the distributed virtual environment (see e.g. [Funk96a]). It influences the number of messages that have to be transmitted in order to keep the database consistent, or to provide determined services (such as directory services, collision detection, etc.) We mainly differentiate between two types of **network architecture**, called client-server and peer-to-peer.

In *client-server* architectures a server is used to mediate the communication between the clients connected to it. As all messages between the various clients pass through one or more server(s), the latter can *explicitly filter* out messages (e.g. by employing visibility culling) or process/enhance the messages, like adding client-specific info, or aggregating multiple messages for the same recipient into one message. Servers can provide *services* like a centralized information directory, collision detection, or database management; hence this structure matches the '*central repository*' (see Section 2.2). Furthermore, if more than one server is present in the environment, they can be arranged in a hierarchical structure, to improve the efficiency of the database management or message transmission.

In a *peer-to-peer* network, the clients exchange their messages directly; the communication between them is not routed through a server. This structure, which matches a '*distributed database*' (Section 2.2), has the advantage of avoiding the additional latency introduced if all communication passes through one or more intermediate server(s). But it also increases the difficulty of keeping the hosts' databases consistent or performing resource-optimizations like Area-of-Interest filtering (see Section 4.1): each sending host must have information about its communication peers, it must know which data the other hosts need. Hence this architecture puts increased size, memory and processing requirements on all hosts, and it is more difficult to locate entities, players or services. Figures 2.2(a) and 2.2(b) show a simplified representation of a client-server and peer-to-peer structure, respectively.

Very often a *hybrid approach*, combining client-server and peer-to-peer approaches, is employed: some clients exchange their messages via a server, and others communicate directly. Additionally, pure peer-to-peer networks (which contain no server at all) may be enhanced by servers responsible only for providing determined centralized services, such as directory services to quickly locate objects or other services.



(a) Client-server architecture. (b) Peer-to-peer architecture.

Figure 2.2: Network architectures.

Other than by the architecture (or physical structure) of the network, the communication is also affected by the **network protocol** employed. The communication between sender and receiver, both in client-server and peer-peer networks, can happen in a *connection-oriented* fashion, which is tailored to the reliable transmission of data streams (such as the *TCP/IP* protocol), or via a best-effort, *connectionless* packet-delivery service like *UDP*.

- **TCP/IP** connections ([Post81]) set up a direct communication channel between sender and receiver. They require an explicit connection establishing and removal, and the continuous management of all open connections requires additional overhead. But TCP/IP provides a reliable communication service and supports continuous data streams.

- The **UDP** protocol ([Post80]) is a connectionless service, where small packets are routed through the network on a best-effort basis. No explicit connection is set up: the communication is rather based on a 'fire-and-forget' philosophy. Once a sender has dismissed some packets, it has no guarantee that they will arrive at their destination uncorrupted, in the correct order, or arrive at all. But in advantage no overhead is required for explicitly opening, managing and closing the connections. Hence UDP allows a much higher number of connections, and is encouraged if a sender frequently changes the receiver of the messages.

TCP/IP connections are *unicast* transmissions (which means, each message is send explicitly from one determined source to one determined destination). UDP connections can be either *unicast*, *broadcast*, or *multicast*.

In a **broadcast** transmission ([Mogu84]), a message sent by a host is automatically delivered to all other hosts in the local network (e.g., all hosts connected to

the ethernet cable). The advantage is that the message must be sent only once, reducing the overhead for the sender. The disadvantage is that all hosts receive the message, and must hence process it, which introduces an overhead for the recipients. Furthermore, the network is flooded with packets. Broadcast transmission mode is advantageous if the messages are likely to be accepted by most receivers; in this case the time required to distribute the messages is considerable lower compared to **unicast** transmission, where each message must be sent to all recipients separately. However, if the message is of concern only to few hosts in the network, the additional overhead for the receivers and the increased bandwidth requirements of broadcast transmission are likely to outweigh the benefits.

Multicast transmission ([Deer89]) tries to overcome this problem by specifying for each message a list of recipients that should receive the message. Hence the sender still has to transmit the message only once, but the number of recipients can be explicitly limited. Multicast transmission has low bandwidth requirements, but it must be supported by the routers; they have to replicate the message in case the recipients are reached via different output channels of the router.

Although all combinations of network structure and network protocol are possible, client-server architectures typically employ TCP/IP, except in cases a client must change server frequently, or an unreliable transmission is acceptable. Peer-peer networks in contrast mostly employ the UDP protocol (mainly used in multicast transmission mode).

Networking is a very complex field, and this section can only provide a very rough introduction. To learn more about the field, refer e.g. to [Come94] or [Stev94].

2.3.2 Scalability Issues

Peer-to-peer networks, in order to be scalable at all, must limit the number of update messages transmitted over the network. Assume every host manages one avatar, we have $N = number\,of\,objects = number\,of\,hosts$. If a host transmits an update to all other hosts every time its avatar moves, the number of messages grows with an $O(N^2)$ effort, impeding any scalability.

To make the system scalable, it is necessary to apply message filters, so that update messages are not sent to every host for every update. Popular approaches include the use of explicit filtering functions (e.g., visibility culling based on precomputed cell-to-cell visibility) or implicit filtering by means of multicast groups.

A very simple form of explicitly performing visibility culling is to send updates only to those peers whose avatar[1] lies in a cell visible to the cell in which the updates occurred (we assume a cell-based subdivision of the environment and limit the culling to a precomputed cell-to-cell visibility). But this requires each host to maintain an up-to-date list of which cells the other avatars reside in. This technique scales beyond the 'send an update to everybody' approach, but still requires to send update messages to all peers whenever an avatar moves into a new cell. As

[1]The position and direction of the avatar is used as viewpoint to present the environment to the user.

infrequent as this might be, the effort is still $O(N^2)$; hence the system is not truly scalable.

If the network supports multicast, the filtering effort may be shifted from the hosts to the network, by assigning a multicast address to each cell and let the objects send updates to the cell they are actually located in; the hosts can then subscribe to the cells within their area of interest. The hosts do not have to maintain lists, but rather joint and leave multicast groups, as their viewpoint moves across cell boundaries. This involves no explicit update transmissions, but implicitly causes the network to generate messages needed to update the routing tables. This additional overhead, even if being only $O(N)$, grows with the number N of connected clients, hence the system is not scalable infinitely.

The only way to make unlimited scalability theoretically possible, is to employ a client-server architecture, where no additional communication between the clients is necessary to maintain the filtering functionality, and where localized communication between avatars in the same area dominates.

Section 4.1 describes the culling techniques mentioned above in more detail.

Chapter 3

Detecting resource bottlenecks

The most influential resource bottlenecks typically encountered in distributed virtual environments are the *network*, the *graphics pipeline*, and the *computational power* (processor cycles).

The performance of a **network** is mainly characterized by its *bandwidth* and *latency* (also referred to as *delay*). The bandwidth describes the amount of data that can be transported simultaneously by the network; it can be imagined like the amount of water that is observed flowing through a tube on any chosen point of its course. The bandwidth, which is determined by the type of hardware used to transmit and transport the data (see e.g. [Stal96]), is measured in Kbps, Mbps or Gbps (kilo-bits, mega-bits or giga-bits per second, respectively); Table 3.1 shows the bandwidth limitations of some frequent network types. The latency in turn describes the time required by one bit of information to travel from the sender to the receiver. Due to its physical nature, a network always introduces some delay, hence every information transmitted will always be outdated to some extent when it is received.

Modem	14.4-56 Kbps
DSL	1.5 Mbps
Cable modem	10 Mbps
Ethernet	10-1000 Mbps
Fiber optics	10 Gbps

Table 3.1: Bandwidth data of connection hardware.

The limitations of the **graphics pipeline**, responsible for rendering the environment and producing a frame rate fast enough to perceive a smooth animation, are usually determined by the graphics card, and expressed in 'polygons per second' (*polygon throughput*) as well as 'pixels per second' (*fill rate*). The number of polygons-per-second a graphics card can support describes the number of polygons that can be processed per unit time (in this case, per second), including geometric transformations and projections. Thus it is a measure of the polygon throughput, which limits the *number* of polygons the objects in the environment can consist of.

The pixels-per-second give the fill rate of the graphics card, describing the number of pixels that can be drawn per second. This affects the *appearance* of the polygons' surface, e.g. the use of a texture.

A typical approach is to first reduce the number of polygons used to represent the environment (e.g. by using Levels of Detail or performing visibility culling), then to limit the number of pixels needed to render the surface of the polygons (e.g. by selecting textures of varying complexity). Table 3.2 gives the performance data of some widely used graphics cards. However, these numbers must be taken with great care as they are based on different evaluations, and some specifications (such as the PS2 and especially the XBox) are synthetic values estimated by the vendors and not based on realistic gaming environments.

Nintendo 64	150K
Sony Playstation	350K
Sega Dreamcast	3M
Gamecube	6M-12M
GeForce2	25M
Sony Playstation 2	75M
XBox	100M
SGI Infinite Reality 2 Multipipe (16)	210M

Table 3.2: Polygons per frame of graphics hardware.

The **computational power** needed to traverse the mainloop depicted in Section 2.1 is the third major bottleneck typically present in a DVE. Processor cycles are needed to query the input devices, read data from and write data to the network, and support the graphics pipeline in the generation of the images necessary to provide a smooth animation. The computational modelling is especially demanding: simulating the motion of the own entities, predicting the motion of the remote entities, performing collision detection, etc.

Furthermore, a substantial amount of processing power is needed to apply resource management techniques, such as reduction/filtering or scheduling algorithms. Hence the effort of decreasing the network and/or graphics bottleneck increases the processing bottleneck; the optimization of one resource is often at the expense of another.

Managing these resource bottlenecks in virtual environments is crucial to the performance, scalability and user acceptance of the system. User studies indicated that people experience a decrease in performance if the entire mainloop takes more than 100 msec ([Wlok95]), hence the resource optimization techniques should prevent the bottlenecks to affect the system beyond that point.

We must be aware that the optimization techniques which have the task to reduce the load on the resource and manage their usage, are themselves consumers of these resources. It is necessary to carefully consider which resources are consumed to optimize the usage of other resources; the optimization of the graphics pipeline or the network is typically at the expense of processor cycles; lowering the load on

the processor often requires to transmit more messages. We must remain within the confines of the available resources and time constraints; the resource optimization techniques employed should on one side reduce the load on the resources and minimize the effect of the bottlenecks, and on the other side allow the system to run in 100 msec loops.

Every new user that participates in the environment increases the demand for resources, thus directly affecting the scalability of a system and the requirement for efficient resource management techniques.

- New users must receive the initial state of the database at startup, and updates from the other objects at runtime. Furthermore, they generate updates themselves that must be sent to the other participants. All these actions put additional load on the network.

- New users also increase the load on the graphics pipeline and the demand for processor cycles, as the objects introduced by them must be rendered, monitored and stored by all other hosts.

Minimizing the demand for resources does not only help to avoid bottlenecks; even if enough resources are at disposition, an efficient management - leading to a decreased demand for resources - will improve the performance of the system: ranging from a faster transmission of data and a better interactive performance to a smoother animation.

Furthermore, it helps alleviate another problem that is introduced by the existence of inhomogeneous components in distributed systems: if the hosts interacting in the common environment differ in their equipment and hardware performance, it is more difficult to maintain a consistent state and a consistent view into the environment for all participating users. It is up to the system designer which strategy to follow: exploiting all resources available or use the lowest common denominator. If every host exploits all of the resources it has at disposition, a high-performance graphics workstation can provide a much higher frame rate and more detailed representation of the environment than a low-cost personal computer. This will favor some users while penalizing others, when interacting with each other - whereby the penalized user is not always the one with the slower hardware. Singhal and Zyda ([Sing99]) stated an example of military exercises based on NPSNET, in which the participants with a simpler and less detailed representation of the battlefield were able to spot the enemies faster than the users which had a highly detailed representation.

If the system tries to find a lowest common denominator, it attempts to give all users an equal chance, and an equal main loop (similar frame rates, response time and representation details). However, parts of the resources will be wasted, and one slow component can massively penalize the whole system.

Chapter 4

Managing network bottlenecks

In Chapter 3 we have presented three major bottlenecks encountered in distributed virtual environments: the *network*, the *graphics pipeline* and the *processing power*. As the evaluation of the PRR algorithm (Chapters 9 and 10) focuses on the optimization of the network bandwidth usage, the next sections deal extensively with network bottlenecks. Section 4.1 examines the various *reduction techniques* typically employed to deal with network restrictions; for the sake of completeness the management of graphics and processing bottlenecks is shortly mentioned in Section 4.2. The *scheduling techniques* and their typical usage in connection with network limitations are briefly overviewed in Section 4.3; Section 4.4 finally depicts how reduction and scheduling techniques can be combined with the help of the PRR algorithm.

4.1 Reduction techniques

A wide range of techniques have been developed in the recent years to deal with the network bottlenecks that typically arise in distributed virtual environments.

The simplest approach is to optimize the **communication protocol**, transmitting only the messages that are strictly necessary. This includes reducing the message size (*compression*), as well as *aggregation* of multiple messages for the same recipient into one big message (NETEFFECT [Das97], for example, employs this concept to perform group dead reckoning). Aggregation does not reduce the amount of data transmitted or the number of recipients, but the total number of messages that are sent. Concerning the compression of messages, other than raw binary compression, more sophisticated approaches include compression of geometry data ([Dans94]), polygonal data ([Deer95]), or a combination of geometry and image data ([Levo95]).

Area-of-Interest (AOI) management techniques are among the most popular approaches to reduce the load on resources. Basically, they try to reduce the amount of information that is generated, processed and transmitted by filtering out the information that is not of interest to the recipient (e.g., the objects which are

invisible). As each host is usually focussed only on a small portion of the environment, the so called 'Area of Interest' (AOI), a substantial amount of resources (especially of network bandwidth required) can be saved by transmitting the host only the data of those entities that can be perceived. Objects which are invisible and hence cannot be perceived by the user, are often preempted in favor of the visible ones. A generic treatment of the AOI notion can be found in the aura-nimbus model described in [Gree97].

AOI techniques are so called '*filtering techniques*'; for the sake of simplicity this term is often used synonymously for all different types of '*reduction techniques*', although only AOI approaches (and, to some extent, Dead Reckoning techniques) perform an explicit filtering function on the objects.

Among the simplest forms of AOI management is to require each host to *explicitly register interest* in the chosen objects. This approach is successful if a host is interested in a small and mostly static set of distinct objects; otherwise, the subscription and unsubscription effort is likely to outweigh the benefits of the filtering.

In **vicinity-based AOI** techniques, the interest in a determined object is given by the distance to the viewer. The environment is typically subdivided in *regions*, and each user gets updates only from those objects which lay in the regions immediately adjacent to its own location. The subdivision can be arbitrary or regular (e.g., employing a regular grid of 2D hexagonal cells, such as in NPSNET [Mace95]). All objects subscribe to the region they are located in, transmitting it their data, such as position updates. This allows all hosts to be informed about the actions in the regions they are interested in, either by monitoring the flow of information towards that region(s), or explicitly querying a sort of region manager.

Visibility-based AOI allows to save network bandwidth by suppressing the transmission of update message which lay outside the field of view of the user. Even if this leads to an inconsistent state of the scene database among the various hosts, this inconsistency cannot be perceived by the users, as it affects only objects which are hidden. The visibility can either be *predetermined*, or specified by a *camera* or *viewcone* at runtime. In SPLINE ([Barr96]), for example, the environment is subdivided into *regions* whose inter-visibility is calculated in advance; the RING system ([Funk95]) employs accurate visibility culling based on dynamic cameras, while AVIARY uses viewcones. In the first case, it suffices to determine which regions are visible from the one the user is actually in, and discard the objects located in the other regions. If the visibility is given by the position, orientation and field of view of a moving camera, in order to make this calculation feasible at runtime, it is usually necessary to subdivide the environment into fine-grained cells and employ so called 'potentially visible sets' (the notion of PVS was first introduced by [Aire90]), which contain the areas of the environment visible from each possible location of the camera. Typically, the environment is triangulated, with lists of potentially visible triangles being used to quickly determine the visibility relationships.

A practical application of PVS can be found e.g. in the UND Walkthrough project described in [Mine95] or [Aire90], but also in our testbed which uses a very simple algorithm to predetermine the PVS (see [Schm97]). A good starting point for a generic introduction into culling is [Clar76]; comprehensive overviews about polygon culling are given in [Mine95] or [Funk96b].

We correlate only the camera-based visibility (as used in RING) to the visibility-based AOI techniques; if the visibility is pre-determined and based on regions (SPLINE, or NETEFFECT), on our opinion it has a greater resemblance to vicinity-based techniques. The reason is that in the latter case the filtering condition is given simply by the presence of the object in determined static regions. However, if the visibility is based on a moving camera, the culling area may be continuously changing and requires a much more fine-grained subdivision of the environment.

Dead reckoning (DR) is another set of techniques that reduce the number of messages transmitted over the network. In dead reckoning, other than simulating its own (local) objects, each host maintains a simplified behavioral model for the remote objects managed by the other hosts. Using this model, it predicts the actual state of these remote objects by extrapolating from the previous states; this operation is referred to as 'dead reckoning'. This technique allows to limit the network transmission to updates messages used to correct the prediction, in case it becomes too inaccurate. To detect this case, every host stores an additional 'ghost'-copy of the own objects simulated locally, to which it applies the same dead reckoning routines as employed by the other hosts for the prediction. Whenever the representations of the (simulated) local objects and the 'ghost' objects differ by a determined threshold, the remote hosts are likely to have the same errors in their dead reckoning, and updates to correct the remote dead reckoning are transmitted.

Although dead reckoning is a generic notion of predicting the behavior of remote objects in order to save network bandwidth, it is typically limited to extrapolating the objects' position based on position/velocity updates (NPSNET [Mace94], for example, employs first-order derivatives, while PARADISE uses the position history [Sing95b]). Exceptions are e.g. the use of dead reckoning for articulated human figures, such as investigated by [Capi97a].

Dead reckoning is limited to a physically based motion of objects that can be computed incrementally from a limited set of values (such as velocity or acceleration for position dead reckoning), in order to allow the remote hosts to perform an accurate prediction. Therefore, as dead reckoning relies on a prediction based on the recent objects' behavior, the fields of application in areas with an unpredictable user behavior, such as in online video games, is very limited. Furthermore, the reduced demand for network bandwidth gained by employing DR goes at the expense of a heavily increased computation effort, as clients have to predict behavior of the remote objects.

Dead reckoning is related to the AOI techniques listed above by the fact that the transmission of updates is determined by the 'amount of interest' of the recipients in the various objects. In DR however, the interest is determined by the

accuracy with which the recipients predict the motion of the objects, rather than by the objects' properties themselves (such as visibility or distance to the camera).

Other ways of reducing the transmission of messages over the network rely on exploiting **user perceptual limitations**. The visual sense is the most important one, but not all information captured by the eyes is effectively perceived. The human brain filters the information, memorizing only the most important parts. Large, fast moving objects, for example, capture the attention more than small, slowly moving objects; motion performed perpendicular to the line of sight is much more conspicuous than motion toward the viewer. Furthermore, the resolution of the output displays and the eyes themselves restricts the amount of detail that can be perceived. Therefore, not all objects laying in the user's field of view are equally important; this allows to prioritize the transmission of update messages.

Levels of Detail (LOD), for example, partially exploit this concept by providing for each object different representations of varying complexity, and repeatedly choose that 'level of detail' which matches the user's perception and the network bandwidth available.

Geometric Levels of Detail provide for the same objects a set of different geometric representations with varying detail, which can range from millions of polygons and high-resolution textures to a flat-shaded representation using only few polygons. Which level of detail to employ is determined e.g. by the distance to the camera: objects which are too far away need a less detailed representation than nearby objects, and are hence represented with a lower LOD. This affects both the number of messages transmitted over the network (objects with many articulated parts, for example, generate more updates than objects with only few moving parts), and the load on the graphics pipeline to render the objects. In more recent virtual environments, much work has gone into providing continuous solutions to the LOD model creation problem. A comprehensive overview about polygonal surface simplification algorithms is given in [Heck97].

However, **Levels of Detail** are not limited to the geometric representation of the objects; they may include areas such as *simulation*, *collision detection*, *physical force modelling* or the *frame rate* itself. LODs can be used e.g. to tune the precision of simulating the objects' movement, their behavior, the collision detection, or the physical force modelling. Objects which are far away from the camera need a less precise animation than nearby ones. The LODs can be used to affect the amount of data that is transmitted: for example only position updates for distant objects, but position, orientation, joint angles, color updates, etc. for nearby objects. Even the frame rate, the frequency with which the update information is sent, might be affected by the LOD of the objects. From this point of view, also the PRR algorithm can be classified as a LOD technique, because the priority assigned to the objects determines the frequency of their update messages. These Levels of Detail are usually implemented by assigning each object multiple independent data channels ([Sing95b] calls them fidelity channels), each providing information at a different LOD, affecting precision and/or frequency ([Kess96]) of the data.

Delaying the delivery and representation of low-priority information is also a possible way to reduce the load on the network or the graphics pipeline, respectively; but it is a rather infrequently used approach. Examples can be found in [Harv97].

The performance of the optimization techniques employed can be largely supported or hindered by the network architecture chosen for the distributed environment (refer to Section 2.3), which has a substantial influence on the network load required.

A client-server architecture, for example, is better suited for complex filtering mechanisms than peer-to-peer networks; all communication is routed through servers which have a 'centralized' knowledge of the clients' objects and can perform an *explicit filtering*, as well as modifying and enhancing the messages (such as aggregation of packets for the same recipient); the disadvantage is that the servers introduce an additional latency in the network transmission. Peer-to-peer networks, in turn, are suited to perform an *implicit filtering* using for example multicast groups. Multicast is only able to perform a coarse filtering, based on a list of 'subscribed' destinations, but allows to shift part of the filtering effort to the routers and network interfaces.

Multicast is very suited for vicinity-based AOI management, because the implicit filtering achieved by multicast-groups is accurate enough for region-based culling: each region is assigned its own multicast address; clients subscribe to the multicast address of the region they are located in (sending their data to that address). Hosts can then receive the updates of all objects in a determined area by listening to the multicast address of the corresponding region(s). Hence for achieving vicinity-based AOI optimization, a peer-to-peer architecture is sufficient.

If visibility-based AOI management determined by a dynamic camera is needed instead, subdividing the environment into regions and providing a multicast address *per region* is not accurate enough; rather a multicast address *per object* is necessary, providing a very fine-grained partitioning. An alternative approach is to employ region-based multicast groups to first achieve a rough implicit filtering, for then performing a fine-grained explicit filtering on the data transmitted in each multicast group; however, this considerably increases the overhead. Employing *per-object* multicast groups requires each host to continuously monitor which objects are actually present in its changing viewcone, in order to be able to subscribe to the multicast-address of those objects which are momentarily visible.

Therefore it is necessary to provide a directory service capable of delivering the actual location of the objects, or return all objects contained in a specific area. In peer-to-peer networks, this service can be implemented in a distributed fashion, to meet the network's architecture: each host is responsible for providing information about its local objects. In client-server networks, where all communication is managed by one or more server(s), this service is best provided by the servers themselves, which can easily monitor the motion of objects and cameras as they forward (and on demand process) the data packets for the clients. Alternatively, a hybrid solution may be employed, where the hosts send information about their

objects to a centralized server, which in turn can be queried by the hosts to obtain the multicast addresses to subscribe to.

The use of multicast groups has several limitations:

- The routers in the network must support multicast addressing. Today there still exists a substantial amount of routers which do not.
- The number of multicast groups that can be supported by network cards and routers is limited, so it may be not possible to assign a dedicated address to each object.
- Subscribing and unsubscribing to multicast groups requires an overhead and introduces some delay; hence these actions should not be performed frequently.

These limitations can often give a client-server architecture an advantage over a peer-to-peer network. For example, if the cameras are moving at fast speed, the set of objects in their viewcones is continuously changing. Using peer-to-peer networks and multicast groups to perform the visibility culling requires to assign a multicast address to each object, and each camera to subscribe to the multicast addresses of the objects in their field of view. This might lead to a subscription overhead much higher than the additional delay introduced by the servers in a client-server architecture. If the camera is highly dynamic, the adaptation (subscription/unsubscription) to the continuously changing set of objects in its viewcone is likely to outweigh the latency advantages of the peer-to-peer networks.

4.2 Using reduction techniques for graphical and processor bottlenecks

The limitations of the **graphics pipeline**, which is responsible for the generation of images at a frame rate fast enough to produce a smooth animation, are basically determined by the *throughput* and *fill rate* of the hardware. Throughput is expressed in number of *polygons* that can be handled per unit time (typically a second); it puts restrictions on the geometric structure of the objects. The fill rate limits the number of *pixels* that can be drawn (per second), thus affecting the representation of the object's surface, e.g. the detail of the textures employed.

The throughput restrictions are typically met by reducing the number of polygons to be drawn. Visibility culling, a filtering technique also employed to improve the performance of the network, can be used to discard the objects invisible to the user. Subsequently, the number of polygons required to display the visible objects is further reduced by employing geometric LODs.

Levels of Detail can be also used to reduce the number of pixels required to display a determined object, e.g. by providing textures with varying resolution used to represent the surfaces of the polygons.

Processor bottlenecks are handled in a similar fashion. The shortage is due to an excessive amount of information that has to be processed. Processor cycles are required by all steps of the main loop (refer to Section 2.1), but especially by the computational modelling (simulation) of the objects. The more objects have to be simulated, and the more precision is used in the simulation computations, the higher is the demand for processing power. Therefore, AOI filtering which discards objects not of concern to the user reduces the load on the processor, e.g. by preventing invisible objects from being simulated. Levels Of Detail in turn can be used to provide various 'simulation LODs', which rely on processing the information at various precision ([Carl97]). The simulation of avatars far away from the user's viewpoint might for example suffice with the translation of static low-polygon objects; avatars in the immediate vicinity of the user might require a physically correct simulation of articulated human figures including the user of inverse kinematics (for an introduction into the realistic representation and animation of virtual human figures see e.g. [Capi97b] or [Pand96]).

Of course this discussion gives only a rough overview about the problem, but as the Priority Round-Robin scheduling algorithm is evaluated by focussing on the network bandwidth optimization, the treatment of graphical and processor bottlenecks is beyond the scope of this thesis and treated here only for completeness.

A fact that must always be considered when dealing with resource bottlenecks is that the various resources in a system are all related to each other; bandwidth and processor implications, for example, parallel each other closely in networked virtual environments. Optimizing the network bandwidth is usually at the expense of an increased demand for processor cycles; speeding up the simulation of the objects by sending accurate background information or performing parallel computing e.g. increases the traffic on the network. Hence it is necessary to evaluate the impact of a resource optimization technique on all resources in a system, and determine the overall cost versus the benefit achieved.

[Sing99] describes the relationship between networking and processing in distributed virtual environments with the so called 'Information Principle Equation':

$$resources \approx M \times H \times B \times T \times P \tag{4.1}$$

where

M	number of messages transmitted
H	average number of destination hosts for each message
B	average network bandwidth required for each message
T	timeliness with which the network must deliver messages (a large number implies minimal delay)
P	number of processor cycles required to receive/process each message

This principle expresses the fact that an accurate compromise between the resource optimization techniques employed and the resources required by these techniques must be found in order to improve the scalability and performance of a system.

As we must stay within the confines of the available resources (network, processor cycles, etc.), the scalability of the system is limited; if an excessive number of objects demands too many resources, the resulting bottlenecks cause a degradation of the system's performance. Employing reduction/filtering techniques to reduce the number of competing objects does not suffice, because in case the remaining load still exceeds the availability of the resources, the bottleneck problem persists, and the degradation of the system's performance it not graceful at all. This problem might be partly alleviated by employing scheduling techniques.

4.3 Scheduling techniques

Scheduling conceptually follows a different approach than reduction (or filtering) techniques. It does not reduce the number of objects competing for a determined resource, but rather tries to select a determined subset among all competing objects (which are granted the resource requested), in order to optimize one or more determined system parameters.

This problem is known from operating systems research, where independent processes competing for CPU power have to be scheduled for assignment of processor cycles; a comprehensive overview can be found in [Tane92], [Stal95], [Deit90] or [Silb94].

The simplest scheduling policy is executing the processes one after another in their order of submission (that is, using their age or time of arrival as priority), called *First Come-First Served (FCFS)*. FCFS is a so called *non-preemptive* algorithm, where a selected process terminates before a new process is scheduled.

In *preemptive* algorithms, the currently running process may be interrupted by another process. This happens for example in *Round Robin (RR)* scheduling, the preemptive version of FCFS, where each process is removed from the resource and reinserted at the end of the queue if it exceeds a determined time slice. As RR is output sensitive and immune to starvation, it is a very popular technique; however due to its inability of enforcing priorities its performance is rather limited.

A scheduling algorithm which employs priorities and also includes a feedback from the system is the so called *Multilevel Feedback Queue (MLFQ)*. It consists of levels with decreasing priorities; the algorithm starts with the highest level and picks all processes present in that level in a round-robin fashion. After a determined time slice the actually selected process is preempted and inserted in the next lower level; thus the priority of the processes decreases with increasing execution time. When a level is empty, the algorithm selects the processes of the next lower level. New processes are inserted in the highest level, a fact that may lead to a starvation of the processes in the lower levels. To overcome this problem, processes waiting to be scheduled may be raised to a higher level after a determined amount of time.

However, the MLFQ is not apt to be employed as generic-purpose scheduling algorithm for virtual environments, as there are substantial differences between the scheduling of processes in operating systems and the scheduling of objects in a virtual environment:

- Processes are usually scheduled *only once* (except rescheduling because of preemption), after which they are terminated and removed from the scheduling queue. For further scheduling, the processes have to be resubmitted to the algorithm, where they are treated as new processes. In virtual environments, we often have to schedule the same object *repeatedly* (e.g. recurrently sending position updates of a moving object).

- If priorities are employed by process scheduling algorithms, they are used to determine which process is to be scheduled *next*: the process with the highest priority is chosen, but this may lead to starvation of lower priority processes (e.g. in the MLFQ). Techniques to avoid starvation employ a constant monitoring of all processes to treat lower level processes (or penalize high level processes); this leads to an overhead depending from the number of processes. In virtual environments, where objects are scheduled repeatedly, the priorities are rather related to the scheduling *frequency* of the objects (the amount of time to wait between two *consecutive* schedulings), as opposed to the waiting time until the *next* scheduling.

- Process scheduling algorithms usually deal with a reasonable number of processes, which allows them to continuously examine *all processes* to determine their characteristics. As the overhead of our algorithm should not depend on the number of objects (this would imply an input-sensitive behavior), it is prevented from sorting or comparing *all objects* against each other.

- The 'amount' of resources required by processes may *vary* substantially, thus it is often necessary to *preempt* the execution of a process to be resumed later. Furthermore, it is necessary to distinguish between CPU and I/O-bound processes. For PRR scheduling we assume all objects need a small, *constant* amount of non-blocking resource (e.g. transmitting an update over the network), so that they can be *serviced completely* when being scheduled.

What most process scheduling techniques have in common with our attempt to schedule objects in virtual environments is the attempt to optimize determined system parameters, to minimize the risk of starvation (every object should be guaranteed to be serviced at least once within a determined amount of time), and to enforce priorities.

4.4 Combining reduction and scheduling techniques

Usually reduction and scheduling techniques are *combined* to achieve better performance. In a strict sense, most reduction/filtering techniques require the additional use of a scheduling algorithm, in case the resulting number of objects still exceeds the availability of the resource. The most widely used scheduling algorithm in this situation is *Round-Robin*, due to its output sensitivity and starvation free performance. However, scheduling techniques select a subset of objects which is granted the resource requested, but this is always only an approximation of the optimum state (all objects getting the resource), as good as the selection may be. Hence the less objects are competing, the better is the approximation that can be made by the scheduling techniques.

The *Priority Round-Robin (PRR)* approach presented in Chapters 6 ff. is not only a very efficient scheduling algorithm, but also allows a better integration of scheduling and reduction techniques than the traditional approach, which is to *first* apply a reduction/filtering technique on the competing objects, and *then* applying a scheduling technique on the remaining ones. PRR *includes* the reduction techniques in the determination of the objects' priorities, without renouncing to an output sensitive behavior (see Chapter 7). An evaluation of the PRR algorithm can be found in Chapters 9 and 10, where it is employed to manage the transmission of update messages in a client-server system. Visibility information will be included in the determination of the objects' priorities when they are scheduled; this allows to preempt the transmission of invisible objects in favor of the visible ones.

Chapter 5

Examples of resource management in virtual environments

Almost all existing virtual environments employ resource management techniques to reduce the load on their resource and minimize the effects of bottlenecks.

SIMNET ([Pope89], [Calv93]) was one of the first existing DVEs; its development started in 1983 and was officially discontinued in 1990. SIMNET was commissioned by the US Department of Defense (DoD) as a military simulator for small unit training. It follows a peer-to-peer architecture; every host replicates the database completely and acts as an autonomous simulation node responsible for managing one or more entities, such as vehicles, planes or soldiers. According to a so called 'object-event paradigm', each node is responsible for placing messages about every state change of its objects on the network, so that they can be perceived by the other nodes. As broadcast is employed to distribute the event messages, the network is continuously flooded with packets; multicast is supported by the system, but only to run multiple independent 'exercises' simultaneously. Although dead reckoning is employed to reduce the number of packets transmitted, due to the elevated network load the SIMNET specifications require the use of dedicated high-performance networks.

SIMNET is a highly proprietary system requiring specialized hardware, thus in order to be further usable by a broader range of researchers, the **DIS** project was conceived in 1990 to generalize and extend the SIMNET protocol, allowing the simulation of different types of players on different types of machines. Formally DIS is a protocol, not an system architecture; it became an IEEE standard in 1993 [IEEE93]. The DIS protocol consists of 27 different types of messages (so called Protocol Data Units, or PDU's) that every node participating in the system must be able to read and write. DIS does not introduces new resource management techniques in addition to those employed by SIMNET (it uses broadcast to distribute

the messages, and support dead reckoning, although with a selection of 9 different types).

Both SIMNET and DIS are military developments contracted by the DoD, but encountered some interest in the academic community, thus in 1990 the Naval Postgraduate School started the development of **NPSNET** ([httpNPS]), aiming at providing a low-cost version of SIMNET. In its first version, NPSNET is limited to read SIMNET databases; DIS compliance was achieved in version IV presented in 1993 ([Mace94]). Additionally to the dead reckoning of SIMNET and DIS, NPSNET IV employs a vicinity-based AOI filtering, based on a subdivision of the environment into 2D hexagonal cells ([Mace95]). Each cell has an associated multicast address, with every object subscribing to the cell it is located in. In this way information about nearby objects can be collected by subscribing to the multicast addresses of the neighboring cells.

Due to their system architecture, SIMNET, DIS and NPSNET support only a limited number of participant; except for a few specialized research implementations (such as [Calv95], which sustained 5000 entities for 3 to 5 minutes), the maximum number of participating entities ranges from approximately 500 to 1000 objects.

Managing a higher number of objects requires the use of further resource management techniques, such as in RING, PARADISE or NETEFFECT; these systems aim at supporting more than 1000 objects. **PARADISE** ([httpPARA]) is a hybrid peer-to-peer system, where 'external' servers provide an entity directory service. Each object registers at the server, and is assigned a multicast address.The various hosts can then subscribe to objects in their vicinity by using the directory service. Furthermore, PARADISE uses an advanced dead reckoning mechanism based on the position history of the objects, and can aggregate nearby objects into groups to optimize the multicast filtering ([Sing95b]).

NETEFFECT ([Das97]) is a client-server architecture, with clients linked to a cluster of servers via TCP/IP connections. Each server manages one or more 'communities' which can be populated by user-controlled avatars, and is responsible for the transmission of update messages between the various clients. NETEFFECT employs 'group dead reckoning', based on visibility predetermined by the system designer: the dead reckoning updates for objects in the same building or room are grouped together to reduce the number of messages transmitted. Furthermore, the communication between the servers is limited by managing all objects in a determined 'community' on the same server, independent from the physical location of the connected clients which manage those objects (the counterpart would be to let each server manage the objects of the clients connected to it, independent from the affiliation of the objects to a community).

The **RING** system ([Funk95]), based on a client-server architecture, reduces the number of messages transmitted by employing a camera-based visibility culling; the communication between the clients is managed by the server(s). The visibility computation is based on a division of the environment into cells, and on sets of potentially visible cells precomputed by the server(s). The latter keep track of

which cell an object is actually located in, and maintain a list of the contained objects for each of the cells. RING also uses 'surrogates', a sort of dead reckoning, to approximate the behavior of the objects. In contrast to NETEFFECT, the connections between clients and server(s) are based on UDP, thus clients can change server frequently without connection overhead.

The **VLNET** system ([Capi97b]), developed to investigate the realistic representation and interaction of virtual human figures in shared virtual environments, tries to minimize the number of update messages required to animate humanoid avatars by employing 'motor functions' [Pand96] and 'human dead reckoning' [Capi97a]. Motor functions provide parameterized motions, based on a dedicated set of motion parameters approximated from biomechanical experiments; human dead reckoning focuses on employing dead reckoning techniques for the motion of humanoid virtual models.

Other academic distributed virtual environments are e.g. **SPLINE** ([Barr96]), **AVIARY** ([Snow94]), **WAVES** ([Kazm93]), **MASSIVE** ([Gree95]), **BRICKNET** ([Sing94], [Sing95a]), **MR-TOOLKIT** ([Shaw93]), **DIVE** ([httpDIVE], [Carl93]) or **DVE** ([Grim91]); however they are not discussed in more detail as they do not employ previously unmentioned resource management techniques.

Chapter 6

Priority Round-Robin Scheduling

6.1 Overview

As anticipated in Section 1.4, the idea of developing Priority Round-Robin (PRR) scheduling originated from the need for a fully scalable filtering technique (visibility culling), to be employed in distributed virtual environments. Visibility information is often employed in DVEs in order to decrease the network load, by transmitting only updates of objects which are visible. It is required to compare the objects in the environment against the viewcone of the cameras visualizing the scene, and discard those objects which lay outside the viewing frustum; this procedure is referred to as 'culling'. A popular approach to implement such filtering functions in DVEs is to employ a client-server architecture, where the server manages the transmission of all update messages between the clients. However, assigning the server the task to perform visibility culling for all connected clients causes a substantial overhead. To transmit each client only updates of the visible objects, it is necessary for the server to keep track of the point of view for every client, and continuously compare all objects against the viewing frustums. Assuming $N = number\ of\ clients = number\ of\ objects$, examining all objects for all clients leads to an effort of $O(N^2)$, which substantially affects the scalability.

The PRR algorithm aims at reducing this effort; it provides a **prioritized** management of the update messages transmitted from server to client, including **visibility culling** in the determination of the messages' priorities. PRR has a constant effort of $O(k)$ per client, where k is the number of updates that have to be selected; this leads to an overall effort of $O(k * N) = O(N)$ for N connected clients. Thus PRR has an **output sensitive** performance, a crucial requirement for the construction of scalable environments.

But Priority Round-Robin scheduling is not limited to client-server architectures; it can be employed in peer-to-peer networks as well. PRR is a **generic-purpose** scheduling technique that enforces priorities based on a **freely definable error metric**, trying to minimize the overall error. Therefore, it can be employed in almost any situation where objects compete for system resources, because a

resource bottleneck always causes a degradation of the system performance that can be measured and hence optimized by the PRR algorithm. By including e.g. visibility culling in the determination of the objects' priorities, PRR can combine scheduling and filtering techniques, while still preserving an effort of $O(k)$ per client. Furthermore, it overcomes the problem that reduction/filtering techniques do not deal with the fact that the remaining objects may still exceed the available resources; PRR fills in this gap.

The inspiration of the Priority Round-Robin algorithm can be found in the short-term process scheduling known from operating system's research, where a set of independent processes is given processor time in order to optimize determined system's parameters. Two of the most widely used algorithms are *Round-Robin (RR)* and the *Multilevel Feedback Queue (MLFQ)*. RR is widely used due to its simplicity, output sensitivity and starvation-free performance, but prevents the use of priorities. The MLFQ does enforce priorities (it consists of a set of levels with decreasing priorities), but has either to deal with the risk of starvation, or must constantly monitor all processes and thus renounce to an output sensitive performance.

The scheduling of processes in operating systems and the scheduling of objects in virtual environments bears some substantial differences, for example the fact that in virtual environments - as opposed to process scheduling - the objects usually need be scheduled repeatedly, and that their high number prevents an efficient examination or sorting of all objects. However, by combining the basic properties of RR and MLFQ, the PRR algorithm inherits the advantages of both, providing an output sensitive and starvation free performance, and at the same time being able to enforce priorities. It is therefore a valid replacement for RR in most circumstances. We will employ PRR in our client-server testbed to schedule position update messages, a task which is usually handled by a simple RR queue.

The priority management of PRR is based on the assumption that if an object is not granted the resource requested, it accumulates error, e.g. visual error. To be useful for scheduling, this error must be modelled as an appropriate error metric (such as deviation in position); the goal of the PRR algorithm is thus to **minimize the cumulative error** over all objects in the environment, called the 'overall error'.

Each object in the algorithm is assigned a so called *'Error Per Unit' (epu)*, which is a prediction of how much the error will increase in a determined time unit[1]. If the error is a deviation in position, then the velocity of an object is a suitable epu.

While the levels are processed in RR order, each level is assigned a priority, which reflects the frequency with which the objects in the different levels are selected; objects with a higher epu have to be scheduled more often than objects with a lower epu. The combination of traversing each level using RR, but with a different priority, gives our algorithm its name - **Priority Round-Robin scheduling**.

[1]The time unit chosen for the epu does not affect the performance of the algorithm

We call the waiting time between two consecutive schedulings the *repetition count (rc)*; it is a measure of the time an object has to wait between two selections and thus a measure for the cumulative error generated by the object until the next scheduling. All objects in level i have the same repetition count rc_i, which determines the scheduling frequency and hence the priority of level i.

Let lev denote the number of levels and ne_i denote the number of objects in level i. If we repeatedly take one object from each level (we traverse all levels at an equal speed of one), the repetition count is simply

$$rc_i = ne_i * lev \tag{6.1}$$

In the example shown in Figure 6.1, objects A and B (level 1) must wait 6 times between two consecutive schedulings, objects C to F in level 2 have a repetition count of 12, and object G is scheduled every 3 times. The more objects in a level, the longer they must wait between two consecutive schedulings. If all levels are of equal length, the repetition count of the objects is the same as if they were scheduled using the RR algorithm (in which case rc = number of objects).

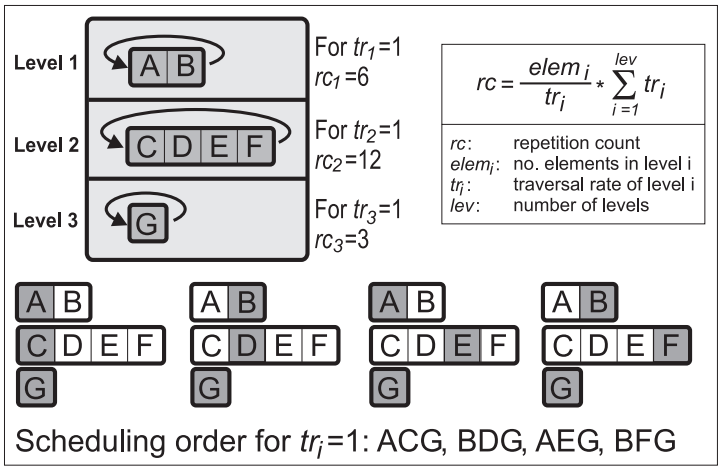


Figure 6.1: Object scheduling order if all levels are traversed at an equal speed of one.

We also see that in the time interval in which the largest level m is traversed exactly once, the other levels i (of equal or smaller size) are traversed at least once. We thus define the *level frequency* lf_i of level i as

$$lf_i = \frac{ne_m}{ne_i} \tag{6.2}$$

Whenever the largest level is traversed exactly once, all objects have been scheduled at least once; those of the largest level one time, and those of the other levels one or more times. The *turnaround time (tt)* in which all objects have been scheduled at least once is simply

$$tt = ne_m * lev \tag{6.3}$$

If the epu of an object can be assumed to be constant (such as for entities travelling at constant speed), a predicted error pe for that object can be calculated from

$$pe = epu * rc \tag{6.4}$$

Furthermore, an estimate of the *total error per level* and the *total error of the environment* can be computed from the epu of each object and the repetition count of the levels. Keeping score of these total error measures is done incrementally with negligible overhead.

6.2 Scheduling for static error distributions

So far the issue of how objects are assigned to levels has not been discussed. Also, it has not been mentioned whether the number of objects in a level is constant or variable. Assuming a constant number of objects for each level, objects in smaller levels get scheduled more often. To fulfill the requirement that objects with a large epu should get scheduled more often in order to minimize the overall error, these objects should be inserted into smaller levels.

If the error distribution of the objects is known a priori, it is possible to fix the number and size of the levels a priori. If we define a set of levels with increasing size and insert the objects with decreasing epu into the levels, then the larger the epu of the object, the smaller is the level (and thus the repetition count) the object is assigned to. After having determined the optimal number and size of the levels from the error distribution of the objects (based on a prediction of the error), we can determine the level to which to assign an object. Each successive level covers a range of possible errors - an error interval - corresponding to the objects it contains. If the error values associated with the objects are completely static, objects will stay in the level they are assigned to.

Unfortunately, dynamic virtual environments do not have a static error distribution. An object's epu will almost certainly change each time it is inspected. Not only must the object then be inserted into another level, but also the error intervals associated with the levels must be adjusted, if the size of the levels is to be kept constant. However, we have found that for large numbers of objects the systems response to these adjustments is slow, and the overall error is often larger than plain RR when object behavior is dynamically changing.

6.3 Scheduling for dynamic error distributions

In order to overcome the aforementioned problem, the size of the levels must be variable. Therefore, the error interval covered by a level is no longer an indicator of where an object should be inserted. We considered two alternative variants of how to assign an object to a level:

- **Minimization of the overall error**: the most suitable level for each object is chosen by estimating for the object's current epu of how the overall error is affected if the object is inserted into each level. The algorithm then selects that level which leads to the lowest overall error. Unfortunately, while this strategy automatically finds the best number of objects for each level, it is not superior to RR: as the assignment to a level is only dependent on global error minimization rather than directly on the epu of the object, this algorithm tends to distribute objects with high and low errors equally on all levels. This leads to levels of equal length and equal average error, with a performance equivalent to RR scheduling. Thus the size of the levels must be variable, and the assignment of an object to a level must directly depend on its epu.

- **Average Error Per Unit**: this approach uses an $average\ epu$ associated with each level to determine the most suitable level for an object. The $average\ epu$ is computed using a moving average. An object is then assigned (according to its epu) to the level with the 'closest' $averageepu$. This does not produce a perfect grouping of the objects according to their error, but does quickly adapt to changing error distributions with no additional overhead.

After some experimentation, the second approach - based on $average\ epu$ - was chosen as the most efficient strategy.

6.4 Optimum traversal rate

In Section 6.1 we have introduced a scheduling strategy that consists of repeatedly picking one object from each level. In this case the average contribution of an object to the overall error is determined by the number of objects in each level (Equations 6.4 and 6.1); the repetition count can only be influenced by level length.

If we assign objects to levels according to their epu, the length of the levels is fixed by the error distribution of the objects. This retains us from minimizing the overall error produced by the objects by determining an appropriate repetition count for each level. The rc cannot be influenced by the scheduling algorithm, as it might contradict to the repetition count fixed by the error distribution.

Therefore we need to determine for each level a different 'speed' with which it is traversed (calculated from the error generated by the level), rather than constantly picking one object from each level. This $traversal\ rate\ tr_i$ describes for each level i the number of objects that are selected from that level each time it is visited (all levels are accessed in turn, as in Figure 6.1). This makes the repetition count depend not only on the number of objects in each level, but allows it to be varied via the traversal rate of the level.

Using a determined traversal rate tr_i, the repetition count rc_i for level i is now given by

$$rc_i = \frac{ne_i}{tr_i} * \sum_{k=1}^{lev} tr_k \tag{6.5}$$

where ne_i is the number of objects in a level and lev the total number of levels. If av_i is the $average\ epu$[2] of level i, we can furthermore calculate a *predicted error* pe_i for level i.

$$pe_i = ne_i * av_i * rc_i$$

$$pe_i = \frac{{ne_i}^2 * av_i}{tr_i} * \sum_{k=1}^{lev} tr_k \tag{6.6}$$

By summing up the errors predicted for each level, we can derive a formula for the $overall\ error\ (err)$.

$$err = \sum_{i=1}^{lev} pe_i$$

$$err = \sum_{i}^{lev} \frac{{ne_i}^2 * av_i}{tr_i} * \sum_{i=1}^{lev} tr_i \tag{6.7}$$

Our goal is to minimize the overall error err by selecting the optimum traversal rate tr_i for each level i. Hence we can build a cost-function err to minimize, with tr_i being the variables of the function. The number of objects ne_i and the average error av_i of each level i can be treated as constant; hence we can ignore the sum of the tr_i in Equation 6.7, and use su_i to substitute for

$$su_i = {ne_i}^2 * av_i \tag{6.8}$$

This allows us to construct the following cost-function err from Equation 6.7:

$$err(tr_i, \ldots, tr_{lev}) = \sum_{i=1}^{lev} su_i * \frac{1}{tr_i} \tag{6.9}$$

This optimization problem is best solved with the help of Lagrange Multipliers: Equation 6.10 allows us to find the extrema of function f, with g being a constraint function for the variables of f:

$$grad\ f = \lambda * grad\ g \tag{6.10}$$

[2] As an object is assigned to a level according to its epu, the $average\ epu$ of a level is the moving average of the objects' epu contained in that level. Simplifying, we can assume that all objects in a level have the same epu, equal to the $average\ epu$ of the level.

Hence we use err as function to minimize (substitute for f) and introduce a constraint function $cons$ (Equation 6.11) given by the sum of all traversal rates, which is assumed to be equal to one.

$$cons = \sum_{i=1}^{lev} tr_i = 1 \tag{6.11}$$

Substituting for function f and g in Equation 6.10 yields Equation 6.12, which allows us to find the values for the variables tr_i where err is a minimum:

$$grad\left(\sum_{i=1}^{lev} su_i * \frac{1}{tr_i}\right) = \lambda * grad\left(\sum_{i=1}^{lev} tr_i\right) \tag{6.12}$$

To do so, we have to build for all variables tr_i the partial derivatives F_{tr_i}:

$$F_{tr_i} : \frac{\partial\left(\sum_{k=1}^{lev} su_k * \frac{1}{tr_k}\right)}{\partial tr_i} = \lambda * \frac{\partial\left(\sum_{k=1}^{lev} tr_k\right)}{\partial tr_i} \tag{6.13}$$

Solving the partial derivatives F_{tr_i} we get

$$F_{tr_i} : -\frac{su_i}{{tr_i}^2} = \lambda * 1$$

and from this we can solve for tr_i:

$$tr_i = -\frac{\sqrt{su_i}}{\sqrt{\lambda}} \tag{6.14}$$

Now, by plugging Equation 6.14 into the constraint function (Equations 6.11 and 6.15)

$$tr_1 + \ldots + tr_{lev} = 1 \tag{6.15}$$

we can solve for λ:

$$\sqrt{\lambda} = \sum_{i=1}^{lev} \sqrt{-su_i}$$

After substituting for $\sqrt{\lambda}$ in Equation 6.14, we get a formula for tr_i:

$$tr_i = \frac{\sqrt{su_i}}{\sum_{k=1}^{lev} \sqrt{su_k}} \tag{6.16}$$

Re-solving for su_i (Equation 6.8) we finally get the optimum value for the traversal rate tr_i, in order to minimize the overall error err.

$$tr_i = \frac{\sqrt{{ne_i}^2 * av_i}}{\sum_{k=1}^{lev} \sqrt{{ne_k}^2 * av_k}} \tag{6.17}$$

The main loop of the PRR algorithm hence consists in simultaneously traversing all levels according to their 'speed' tr_i. Every time an object is selected, it is granted the resource requested (for example transmitting a position update), after which the object is reevaluated: first a new epu is determined (based e.g. on the actual velocity of the object), then it is reassigned to one of the levels according to its epu. Assigning the object to the level whose $average\ epu$ is most close to the epu of the object yields an simple yet effective adaptation to even rapidly changing error distributions. Afterwards, the traversal rate of the levels is modified to account for the new error distribution.

Assuming a fixed number of levels keeps the effort needed to schedule an object constant, so the PRR algorithm can achieve output sensitive behavior. The freely definable epu allows us to include visibility information in determining an object's priority.

Chapter 7

Using visibility information

Visibility information is already available in many existing virtual environments and networked games, usually employed to limit the amount of data transmitted over the network. In indoor scenes, rooms and buildings occlude most parts of the environment; in outdoor scenes the visibility is often limited by a radius around the user, e.g. the so called 'fog of war' in strategy games.

Visibility culling of objects in a virtual environment can be accomplished by first determining the visible area that can be seen from the viewpoint, and then checking which objects are inside and outside that area. Figure 7.1 depicts the visible area for a client, with objects A and B being visible, and object C being invisible.

Usually visibility culling is first used to reduce the number of objects, then a plain FIFO or RR queue is used to schedule the remaining objects; thus the visibility information is employed to insert or remove objects from the queue.

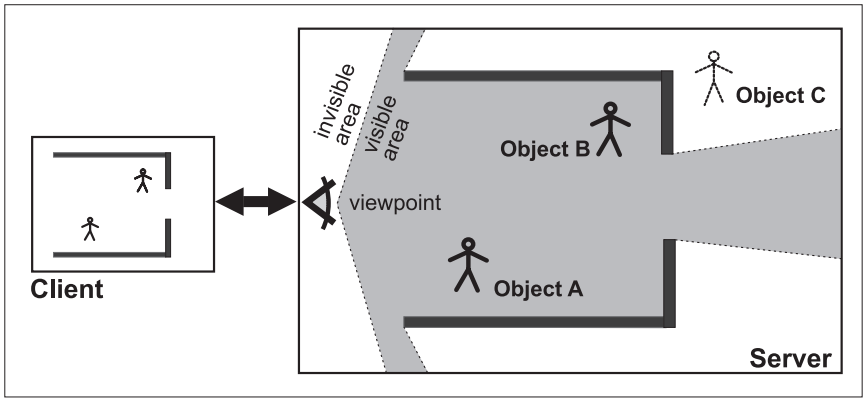


Figure 7.1: Visible area and visible objects of the client's viewpoint.

We replace RR with a Priority Round-Robin (PRR) scheduler and include visibility information in the determination of the objects' priorities. This allows us to reduce the effort for the server to determine which updates should be sent to each

client. As each client has its own field of view, the server must usually examine all objects for each client. Assuming the number of clients approaching the number of objects ($N = number\,of\,clients = number\,of\,objects$), it is an effort of $O(N^2)$.

By employing the PRR algorithm it is possible to shift part of this effort to the scheduler; we let PRR repeatedly schedule as many objects (k) as the network permits, achieving an overall effort of $O(k * N) = O(N)$ for N connected clients. Whenever an object is selected, PRR checks whether it is visible or not. For a visible object the update is transmitted, otherwise the algorithm continues its selection, looking for visible objects, with the highest speed permitted by the computing power and the network bandwidth. The visibility information affects how the object's priority is determined: visible objects get a priority equal to their velocity (their epu); if an object is invisible, the priority is chosen such as to let the object be rescheduled when it is expected to become visible again. In our implementation we base the prediction of when an object will be visible again on the shortest path from the actual position to the next visible area (other than the velocity of the object).

7.1 Temporal bounding volumes

The determination of the time interval an object is supposed to remain invisible is based on a technique called *'Temporal Bounding Volumes' (TBV)*. A TBV is a region of space (for simplicity often a circle or sphere) which completely contains an object for a specific period of time, called the *validity interval* (see e.g. [Suda96]). The TBV becomes invalid if the object leaves the volume. Hence its 'expiration date' is determined by the movement of the object (e.g. rotating around a fixed point, travelling along a track, or translating freely in space) and by the size the TBV can have. In the extreme, a TBV encompassing the whole area of movement of the object will always be valid.

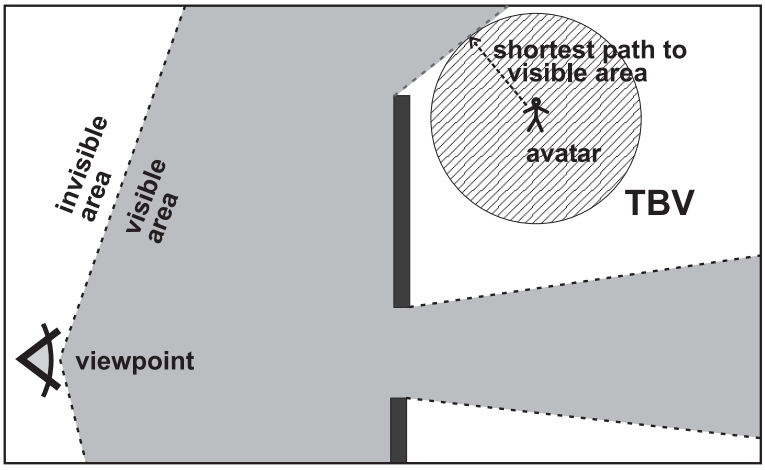


Figure 7.2: TBV for an invisible object based on the shortest path to the next visible area.

For objects with unconstrained translational movement, the expiration date of the TBV is directly related to its size. The validity interval of a TBV could be calculated by dividing the size of the TBV by the velocity of the object. However, in large virtual environments the entities are usually avatars with an unpredictable behavior.

Our application of the TBV consists in using them to determine the priority of objects in the PRR algorithm: every time an object is scheduled, PRR determines whether it is visible or not. In the latter case, a TBV is constructed, based on the time the object is supposed to become visible again (thus, the size of the TBV determines its validity interval). Given the fact that the scheduling frequency of an object is reflected by its priority, we assign the object a priority such as to become scheduled again at the same moment the TBV expires (and the object is supposed to become visible again); this provides kind of an automated 'wake-up' function. Figure 7.2 shows the TBV for an object with unbound translation, calculated from the shortest path to the next visible area (hatched area).

7.2 Integrating visibility information in PRR

In order to be usable by the PRR algorithm, we express the time interval an object has to wait (given by the TBV) in number of scheduling actions[1]. Hence we can directly compare the waiting time of an object - given by a number of scheduling actions - to the scheduling frequency of each level (given by the repetition count, as calculated using Equation 6.5). An object is then assigned to that level whose scheduling frequency best matches its required waiting time.

This causes a difference in how an object is assigned to a level, depending on whether it is visible or not: if an object is visible, it is assigned to that level whose $average\ epu$ best matches the epu of the object (determined by its velocity). If it is invisible, that level is chosen whose scheduling frequency best matches the waiting time determined by the TBV. In the latter case, the epu of the object is not determined by its velocity; rather it temporarily assumes the $average\ epu$ of the assigned level. This allows the PRR algorithm to simultaneously process visible and invisible objects.

[1]This value depends on the number of objects the PRR can schedule per unit time.

Chapter 8

Activity monitoring

One possible origin of errors in the scheduling is an unpredictable or rapidly changing behavior of the objects. The PRR algorithm usually computes the epu of an object based on its recent simulation behavior; but if the object suddenly changes its behavior by a noticeable amount, then the epu that was computed for the object when it was last inserted into a level is no longer valid. The object would need a new epu, but this can happen only when it is scheduled the next time.

Hence in the time interval between the change in behavior and the next scheduling of the object, the priorities and traversal rates as used by the PRR algorithm are not correct. In the worst case, this may lead to an overall error which is higher than that produced by plain RR scheduling. The scheduling frequency of the objects (given by the repetition count of the level they were inserted in) is determined by the relation of their epus; objects with a higher epu get a higher scheduling frequency (a bigger share of the resource) at the expense of objects with a lower epu. For example, an object ranked high in relation to the other objects concerning its epu may suddenly slow down and produce an error (per unit) much lower than most other objects. But until it is rescheduled, it is bound to the fast level it was assigned to, at the expense of other objects which were previously slower, but are now faster (in relation of their epu, alias velocity). Even worse, objects rated low and assigned to a slow level, is denied a higher scheduling frequency until they are rescheduled, in case they should experience a sudden speedup.

If such changes in behavior follow specific patterns, the PRR algorithm can take them into consideration by analyzing the history of the objects; but if the behavior is unpredictable, as occurs very often for human-controlled avatars in virtual environments, the efficiency of PRR is endangered.

To prevent this case from happening, we first need to develop a measure for the 'activity' of objects, in order to quantify the frequency and 'amount' of changes in the epu of an object (reflected from changes in its behavior), and the impact of those changes on the performance of PRR.

8.1 Activity detection methods

Two detection methods were taken into consideration:

Compare the overall error: this method compares the *overall error* generated by *PRR* to the *(predicted) overall error* that would have been generated if the objects would have been scheduled using plain *Round-Robin*. The overall error generated by PRR is given by

$$errorPRR = (\sum_{i=1}^{noElements} epu_i * rc_i) \tag{8.1}$$

while the error generated by RR is described by Equation 8.2 (using RR, the $repetition\ count\ rc_i$ is equal to the number of objects in the level).

$$errorRR = (\sum_{i=1}^{noElements} epu_i) * noElements \tag{8.2}$$

If *errorPrr* is greater than *errorRR*, a response strategy must be triggered.

Compare error and benefit: the second approach is to compare the *predicted error* caused by the behavior changes to the *predicted benefit* achieved by using PRR instead of RR. Every time an object is scheduled (and thus a new epu is computed), the change between the new and the old epu, multiplied by the repetition count between the last two schedulings, is taken as error caused by the change in behavior. This assumes the worst case, namely that the change in epu occurred immediately after the object was assigned to the level. These errors, which include increases as well as decreases in the epu are continuously summed up in a moving average, called the *'error penalty'*. We also experimented with an error penalty that examines only the objects whose epu increased between two consecutive schedulings (hence they were bound too long in a level which became too slow for them); this assumes that the effect of objects whose epu decreased - because they use a 'fast' level at the expense of other objects that might need it - is detected indirectly when examining these objects that were denied the fast level. The latter approach proved to be less reliable than the prior one, and was thus discarded.

The predicted benefit of using PRR instead of RR assumes the best case, namely the difference in the overall error (between PRR and RR) that would have been experienced if all epus had remained unchanged; this is called the *'error benefit'*.

$$penalty = rc_i * abs(epu_i * epuOld_i) \tag{8.3}$$

$$benefit = (noElem * \sum_{i=1}^{noElem} epu_i) - (\sum_{i=1}^{noElem} rc_i * epu_i) \tag{8.4}$$

Equations 8.3 and 8.4 show the formula for the error penalty and error benefit; $noElem$ is the total number of objects in the environment, epu_i and rc_i denote the epu and the repetition count of object i, respectively, and $epuOld$ is the epu the

object had when it was scheduling the last time. Whenever the $error\ penalty$ is higher than the $error\ benefit$ for a determined amount of time (called the monitoring period), the 'behavior threshold' is triggered.

This heuristic proved to be the most reliable for very dynamic objects and was thus adopted as default detection method.

8.2 Activity response strategies

If the behavior of the objects is classified as too instable to rely on priorities for the scheduling, it is necessary to limit the influence of the priorities. In this case, the following response strategies have been tested:

- **Switching**: simply switch to RR performance, hence ignoring the priorities assigned to the objects. All levels are traversed at such a speed as to give the objects the same $repetition\ count$ they would get in plain RR. This produces some undesirable peaks in the overall error when switching between PRR and RR performance.

- **Damping**: specify a maximum difference between the traversal rates of the various levels, thus limiting the influence of the priorities (epu). We divide the interval covered by the $average\ epu$ of all levels into segments of equal length, same in number to the levels in the PRR (this is the highest 'stage' of damping); the length of the resulting segments is then used as maximum difference by which the $average\ epu$ of the various levels is allowed to vary. The nearer the $average\ epu$s are brought together, the more PRR approaches RR performance.

Damping is a heuristic approach (as it is the detection method), but produces good results and a smooth transition between the various stages. Whenever the $error\ penalty$ is higher than the $error\ benefit$ for the monitoring period, the highest amount of damping ($number\ of\ segments\ =\ number\ of\ levels$) is applied; every time that for the duration of the monitoring period the $error\ penalty$ stays below the $error\ benefit$, the stage of the difference-restriction is decreased by one (the number of segments is decreased, until having just one segment left; the latter does not pose any restrictions on the variance of the epu anymore). This leads to smoother transition than just differentiating between 'full' damping ($number\ of\ segments\ =\ number\ of\ levels$) and no damping.

Damping works independently of whether visibility is used in the PRR or not; it allows the PRR algorithm to become a safe scheduling strategy that can cope with almost any error distribution and object behavior.

8.3 Additional damping methods

However, an unpredictable object behavior is not the only possible origin of errors in PRR scheduling. The fact that the speed with which the various levels are traversed is not based on *absolute* values, but rather on the *relation* of the objects' priorities between each other, bears another source of inaccuracy. A PRR algorithm consisting of two levels with an $average\ epu$ of 10 and 1, for example, will traverse the levels at the same speed as another PRR algorithm whose levels have an $average\ epu$ of 1 and 0.1 (assuming the number of elements in each level is equal in both algorithms). Therefore, the scheduling frequency of a level depends not only on the epus and TBVs of the objects it contains, but is affected also by the priorities of the objects contained in the other levels. For a determined level, raising the $average\ epu$ of the other levels or lowering its own $average\ epu$ has the same effect: the traversal speed decreases. Viceversa, if the objects contained in the other levels decrease their epu, the traversal speed increases (same as if its own $average\ epu$ raises).

This leads to the situation that not only every change of an object's priority, but also the assignment of an object to a different level has a determined influence on the whole system; varying the number of objects in the levels affects their traversal rate (even if the $average\ epu$s of the levels do not change). If visibility information is employed, objects change level rather frequently, even if the epus of the objects do not vary: if an object is invisible, it is assigned to that level whose scheduling frequency best matches the validity interval of the object's TBV, while the same object (if visible) is assigned to the level with the closest epu. Therefore it is likely that every transition from a visible to an invisible portion of the environment (and viceversa) causes the reassignment of an object to a different level.

Employing visibility information intensifies two possible sources of errors that must be dealt with. First, the same object may be assigned to levels with very different speeds, depending on whether it is visible or not. It may happen that an object suffices with a slow level when being in an invisible area of the environment (the shortest distance to the next visible area is very long), but requires a fast level when becoming visible (it also has an elevated velocity). If the object still waits in the slow level when becoming visible, the visual error may experience a sudden increase, because the object is now located in a level which might be far too slow, and furthermore the contribution of the object to the overall visual error was suspended while it was invisible (refer to Section 9.5).

Second, invisible objects are generally more prone to variations of the levels' speed than are visible objects. The scheduling of invisible objects relies on a specific 'deadline' that must be met (the moment the invisible object is supposed to become visible, as expressed by the $validityinterval$ of the TBV). Hence invisible objects depend on the *absolute* value of the scheduling frequency of the level they are assigned to; any variation in the level's speed will cause the invisible object to be re-scheduled either too early (before it becomes visible) or too late (after it became visible).

Visible objects, which are assigned to the levels according to their epu, do not rely on a determined speed of the levels. As long as the scheduling frequency of the levels reflects the *relation* of the objects' epu (a high epu require a fast level, a low epu requires a slow level), the PRR will still minimize the overall visual error.

To deal with these problems we include the scheduling frequency - hence, the speed - that was 'promised' to the various objects (when they were inserted into a determined level) in the PRR algorithm. The 'promised speed' is simply the actual traversal speed the level had when the object was inserted (and on which the object's TBV relies). From the promised speed of the objects contained we compute an *average promised speed* for each level, which is now used for the traversal of the levels. This decreases the responsiveness of PRR only marginally, but allows it to handle even rapidly changing visibility situations.

Chapter 9

Evaluation of the PRR algorithm

9.1 Testbed architecture

We will evaluate the performance of the Priority Round-Robin algorithm in a client-server architecture, where it is employed to schedule the transmission of update messages from the server to the connected clients. Although the PRR algorithm can be employed in a peer-to-peer network as well, we have chosen a client-server environment to evaluate PRR for several reasons.

Other than theoretically allowing unlimited scalability (see SubSection 2.3.2), client-server architectures are better suited to perform complex filtering functions, as all communication is routed through the server(s). This automatically gives them access to all information needed to perform the filtering, so that e.g. performing an accurate culling on a viewing frustum determined by the field of view of a camera does not increase the communication overhead compared to a cell-based filtering, which is very popular in peer-to-peer networks. Implementing correct visibility culling in a peer-to-peer environment would substantially increase the communication overhead required to perform the filtering, other than perhaps stressing many clients beyond their hardware capacities. In client-server architectures the clients must perform very little processing or storage to manage the transmission of the update messages; the whole burden of forwarding the messages and performing the filtering functions is assigned to the server(s). Server are typically better equipped than clients, and by employing the PRR algorithm the server can manage the transmission of the update messages - including visibility culling - at a constant effort per connected client.

Another aspect of preferring a client-server architecture for evaluating PRR lies in the priorities of the algorithm itself. PRR enforces priorities in the scheduling of the objects, based on their $ErrorPerUnit$ (we employ the velocity of the objects as epu). As the scheduling frequencies of the various levels relies on a comparison of the objects' epu, each PRR scheduler must have the velocities of all objects at disposition. In peer-to-peer network, this requires additional transmission when-

ever the priority of an object changes. However, the performance increase of PRR outweighs this additional overhead.

And last, but not least, our testbed tries to resemble large scale online games such as Ultima Online ([httpUO]) or Everquest ([httpEVER]), which typically employ a client-server architecture.

9.2 Task of the PRR algorithm

In this setup, our server simulates the movement of a large number of avatars through the environment, trying to roughly resemble the behavior of the avatars in the online games mentioned above. Each of the connected clients can maneuver a camera through the environment, to observe the whole scenery from a dynamically changing viewpoint. As the scene description is transferred to the clients at startup, the communication at runtime can be limited to the transmission of update messages from the server to the clients.

In most existing environments the transmission of updates to each client is managed using a simple RR or FIFO queue, which is easy to implement and has an *output sensitive* effort, crucial for the scalability of such systems. However, if the number of updates to send to each client exceeds the network bandwidth (which might often be the case, even if visibility culling is used to filter out updates of invisible objects), the resulting resource bottleneck leads to a performance degradation causing a database inconsistency (and hence a visual error determined by the objects' displacement).

In our implementation, we manage the transmission of the update messages with a *Priority Round-Robin* algorithm instead of a simple *Round-Robin* queue. Therefore we can schedule the objects according to priorities which are determined from the visual error of the objects (difference in position of the same object on server and client). This not only minimizes the overall visual error caused by the network bottleneck, but also leads to a graceful degradation of the system's performance.

Furthermore, the PRR algorithm can include visibility information in the determination of the objects' priorities. Visibility information is available in most virtual environments, determined by the viewpoint of the various clients and used to filter out the transmission of updates for invisible objects. As we employ a client-server architecture, the server can easily keep track of the position of the objects in the environment as well as the clients' viewpoints; this visibility information is then used when determining the priority of the objects. In contrast to *first* culling the invisible objects, and *then* scheduling the remaining objects (as is the usual approach if a plain Round-Robin queue is used), we let the scheduler manage *all* objects and *include* the visibility information in the priority of the objects. As the determination of an object's priority happens every time and only when the object is selected by the scheduler, this does not affect the output sensitive performance of PRR. Hence the Priority Round-Robin algorithm provides a tool that considerably

eases the construction of scalable environments; it can manage very large numbers of objects and still provide a graceful degradation of the system's performance in case of bottlenecks.

Section 9.3 first describes our testbed implementation, followed by a description of the behavior used to simulate the motion of the avatars (Section 9.4). The error metric employed in the performance comparison is depicted in Section 9.5. The performance increase that can be achieved by replacing plain RR with the PRR algorithm is finally presented in Chapter 10.

We will employ several different system configurations, with and without the use of visibility culling, as well several different behavior models of the objects.

9.3 Testbed implementation

We tried to construct the testbed in such a way as to roughly resemble typical large scale online games (such as Ultima Online or Everquest). Our system consists of a central server managing and simulating an extended environment that can be explored by the connected clients. However, we keep the simulation of the environment very simple. The scenery was generated by randomly replicating a small tile containing a set of walls, and then triangulating the floorplan. It is populated with rigid avatars (represented by simple dots in the screenshots) that are translated through the environment, following some very simple behavioral rules which are depicted in Section 9.4; the movement of the avatars is limited to simple translational displacements. The clients replicate the environment by storing the last known position for each object, thus requiring an update message every time the server changes the position of an avatar. As our implementation aims only at evaluating the performance of PRR, compared to plain RR, the clients need not render the environment.

The server is implemented in a single-threaded fashion; its main loop consists in first translating all objects, for then transmitting each client as many position update messages as the network permits. If the bandwidth permits to transmit only a subset of the pending update messages, a client experiences a visual error given by the difference between the object's position as simulated by the server and the last position update received by the client.

We evaluate the performance of PRR by comparing the *visual error* a client experiences when the transmission of the update messages is managed by a *PRR* algorithm, as opposed to employing a simple *RR* queue. Hence the server manages two different data structures for each connected client, one PRR and one RR queue, filled with all objects that generate position updates. Both queues are traversed simultaneously, transmitting as many updates to the client as the network permits. Therefore in our testbed implementation each client receives two different sets of updates, one from the RR queue, and one from the PRR queue, as illustrated in Figure 9.1; both algorithms is given the same bandwidth, hence both schedulers transmit updates at the same frequency. If the updates provided by PRR are used,

the avatar position (as assumed by the client) is depicted by the green objects, while the red objects represent the RR queue.

To compare PRR and RR, a client stores for each object the last position update received from both schedulers. The visual error is then calculated by summing up (for all objects) the difference between the stored update and the position simulated by the server. In Chapter 10 the performance of PRR is expressed by the percentage the *visual error* of PRR is lower than that of RR; e.g. a value of 50% means the overall visual error can be halved by replacing RR with PRR.

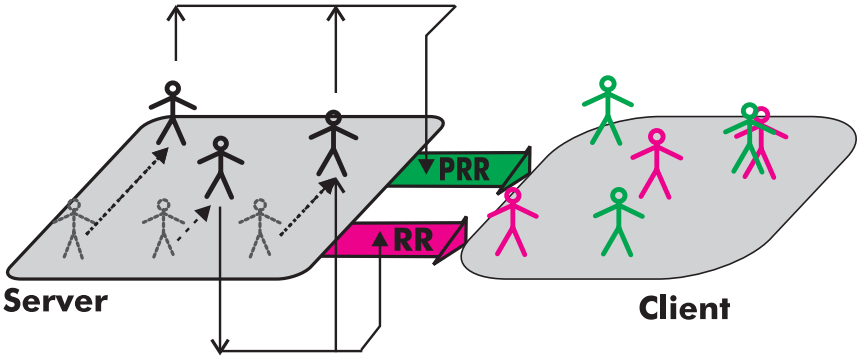


Figure 9.1: Updating the object's position using updates from the RR and PRR scheduler.

Our testbed was implemented to run entirely on a single workstation, hence the network bandwidth limitations are only *simulated*. Although this makes the results more 'experimental' than sending the update messages over a real network, it eases a precise comparison between PRR and RR performance. The transmission of update messages can be 'simulated' by writing the objects' new position directly into a memory region shared with the client, and the position difference can be evaluated immediately with two memory accesses (server memory, client memory).

If we employ a real network for the evaluation, the clients must generate a log file that contains the actual position of their objects for the whole duration of the evaluation. Upon completion, the server can then calculate the position differences using the clients' logfiles and a logfile generate by the simulator. However, this requires huge amounts of diskspace, and the frequent filesystem accesses generate considerable overhead, consuming processor cycles otherwise available to the scheduling algorithms and the simulator. It is possible to avoid the usage of logfiles, but at the expense of transmitting a massive amount of additional evaluation messages, in which the clients report the actual position of their objects to the server. This in turn consumes a substantial amount of network bandwidth, falsifying the network properties the testbed is supposed to evaluate.

As the evaluation of the PRR algorithm is based on network bandwidth and does not consider latency effects, we can assume that an update is perceived by the client immediately after it was sent by the server. This allows us to replace the real network, characterized mainly by a delayed transmission of information,

with a shared memory data exchange; the bandwidth restrictions can be simulated precisely concerning memory accesses as well. Furthermore, our testbed structure allows to simulate a much larger number of objects and clients than a 'real-network' solution would permit.

Thus the single-threaded mainloop of the testbed is implemented as follows:

1. The server translates all objects in the environment by a determined position displacement; this is called a '**simulation step**'.

2. For each client, a limited number of **updates** (only a subset of the total number of objects, in order to simulate the network bandwidth restrictions) is transmitted: first by employing a *Round-Robin* queue, which simply traverses a list containing all objects, and then using a *Priority Round-Robin* algorithm. The number of updates determines the severity of the bandwidth bottleneck (or vice versa); the higher the bottleneck, the smaller the number of updates allowed per loop.

3. The **visual error** for both the RR and PRR algorithm is then evaluated for each client, by comparing the object's position as simulated by the server to the last position update transmitted to the client. First the clients sum up the individual visual errors of their objects, then the clients' errors are summed up into the *overall visual error* encompassing all clients. This calculation is performed for RR and PRR separately, after which the overall error of both is compared.

4. The loop is repeated with step 1.

To make this 'simulated' network as realistic as possible, the number of updates that are transmitted in each loop is based on the specifications of a common hardware equipment. Assuming that most online game are played over the internet, and cable modems or DSL are still rare goods in many areas, we specify the connection between server and clients to be a standard 56Kbps (kilo-bits per second) modem, which means 56.000 bits or 7000 bytes per second. Client-server setups allow each client a dedicated connection to the server, thus we trust a steady flow of 56Kbps between server and client.

We furthermore assume commercial online game providers have enough hardware resources to generate a smooth animation, which requires a minimum frame rate of 24 fps (frames per second); hence the simulator will produce 24 position updates per second for each object. The minimum size of an update packet in our environment is six bytes: two bytes for an object identifier, and other two bytes for the x- and y-coordinate each. As such packets must be transmitted 24 times a second, the 56 Kbps of the connection limits the number of updated objects to approximately 50 ($\frac{7000}{24*6} = 48.61$).

In the examples presented in Chapter 10 we populate the environment with 1000 objects; thus both the RR and PRR algorithm are allowed to update only 50

out of the 1000 simulated on each loop, which is 5% of the total number of objects. This is our reference bandwidth; but to show how the PRR reacts to different network bottlenecks, we repeat all examples with a slightly increased bandwidth, where we allow both schedulers to update 100 objects (10%) after each simulation step.

In the first part of the evaluation we ignore camera position and visibility information. Thus all objects selected by RR and PRR are effectively transmitted to the corresponding client. The second part includes visibility culling: all clients move an independent camera through the environment, whose position, orientation and field of view determines which objects are visible and which ones lay outside the area of interest. As clients perceive only the objects inside the viewcone of their camera, the visual error is limited to the visible objects; entities which are invisible do not contribute to the overall error, as they cannot be perceived by the user (for a more detailed explanation refer to Section 9.5).

Hence it is possible to reduce the visual error by preempting the transmission of invisible objects in favor of the visible ones; the RR and PRR schedulers pass only the visible objects to the network layer, skipping the invisible objects. However, this might require to examine more objects than are effectively transmitted; the amount of time at disposition for this search is limited by the update rate of the simulator. For example, a frequency of 24 position displacements per second means one simulation step happens approximately every 40 milliseconds. This is the amount of time RR and PRR have at disposition to select the updates for their client, before the simulator produces new updates, making the old ones outdated. As we want to make a precise comparison between RR and PRR, a premise of our system is that the server grants each RR and PRR scheduler 40 milliseconds of time to select the 50 (or respectively 100) update objects. We noticed that the overhead introduced by the error measurement functions in our testbed does not allow a homogeneous distribution of time slices; thus we limit the time slices for the schedulers artificially. Our hardware allows us to examine approximately 100 objects on each loop, in order to find the 50 visible objects that can be transmitted. To allow an accurate comparison between the examples in which we schedule 50 objects on each loop, and those where 100 objects are selected, in the latter case we assume our system was upgraded to a faster CPU, and permit PRR and RR to examine 200 objects (to find 100 visible ones).

The PRR algorithm uses the TBV approximation introduced in Section 7.1 to treat invisible objects. This automated 'wake-up' function minimizes the selection of invisible objects; RR lacks such an optimization. Although RR has the theoretical advantage of not being required to perform TBV calculations for its objects, the time needed for this process can be minimized by precomputing a list which stores the distance between all triangles in the floorplan. As each entry requires only one length value, and this list is valid for all PRR algorithms managed by the server, it is a memory requirement feasible by most systems. Using the object's actual triangle as reference, this list allows us to find the nearest visible triangle with few memory lookups.

9.4 Implementation of the behavioral model

The behavioral model with which the server moves all objects through the environment roughly resembles an average conduct profile of avatars in large scale online games such as Ultima Online or Everquest. It consists in selecting a destination, moving to that location with a predefined velocity (the movement consists of a simple translation), and upon arrival repeating the whole process. The environment can be equipped with so called *'hotspots'*, dedicated areas with resemble prioritized haunts (such as rooms, buildings, special places, etc.) frequently present in large scale virtual environments. These preferred whereabouts lead to the situation that the avatars are not distributed equally over the whole environment, but rather tend to group in specific places. Furthermore, avatars do not always rush from one place to another, but rather spend some time in their chosen haunt, before continuing to their next destination. Hence we equip all objects in our simulation with a small set of behavioral parameters:

- Each object has a determined **speed** with which it moves through the environment; this velocity is used as $Error\,Per\,Unit\ (epu)$ when determining the object's priority.
- An individual predefined percentage value is assigned to each object and used to decide between contenting itself with a random location (anywhere in the environment) or requiring a **hotspot**, when selecting a new destination; this allows us to specify the 'attractiveness' of the hotspots and enforce their usage. If the avatar is located in a hotspot when selecting a new destination, we further distinguish between forcing the transition to a different hotspot, or choosing a different position in the same hotspot, in order to spend some time in that area; this resembles the exploration of a determined haunt.
- Additionally, we support the formation of **crowds** in the environment; each object can perform an independent movement, or be instructed to follow the movement of another object. This leads to the migration of groups through the environment.

As the environment consists of a triangulated floorplan, the selection of a new destination is based on triangles. First the shortest path from the actual position to the destination triangle is determined, using a connection graph that was build when initializing the environment. Then a spline is generated, using a random position inside each triangle along the computed migration path as control points. The motion of the objects thus consists in a translation along splines with their predetermined velocity. Once arrived at destination (reaching the end of the spline), the whole process is repeated.

9.5 Error measurement

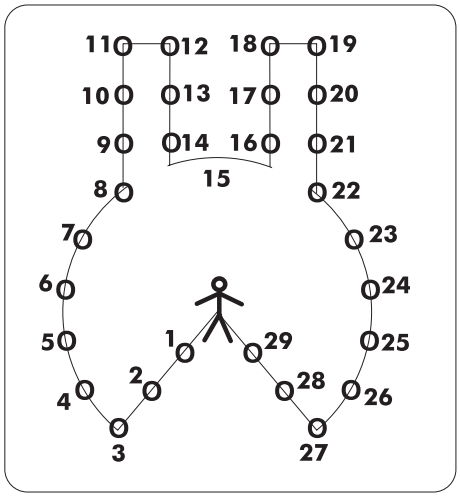
The error metric used to compare the performance of *Priority Round-Robin* and plain *Round-Robin* is based on the visual error given by the objects' displacement. Every time the simulator changes the position of an object, it results in a visual error given by the difference between the position simulated by the server and the last position update received by the client; upon transmission of a position update, the visual error of the object is set to zero. The sum of the objects' individual errors gives the *visual error* of the client; the errors of all clients taken together are denoted as the *overall visual error*.

In order to make the visual error independent from the movement direction of the object, we do not measure the absolute (Euclidian) distance between the object's position (on server and client), but rather perform an incremental computation of the visual error by repeatedly adding the translation *offset* after each simulation step. If we simply compare the object's position on server and client to determine the visual error, a simulated motion which heads away from the last position stored by the client leads to a steadily increasing visual error; but if the client then changes its direction and moves toward the stored position, the visual error would begin to decrease, although the client does not perceive the motion of the object, hence logically increasing the inconsistency. The simple 'Euclidian' metric gives a snapshot of the 'actual' visual error at a determined moment in time, but does not account for any motion performed between two evaluations of the visual error, if the simulated position is the same at both times. Figure 9.2(c) shows the visual error as given by the Euclidian distance if the object performs the motion depicted in Figure 9.2(a). We see that at step 8 and 22 the visual error is the same, although there was a substantial motion inbetween; furthermore, while the object performs a circular motion, the visual error does not increase, although information about the movement is lost.

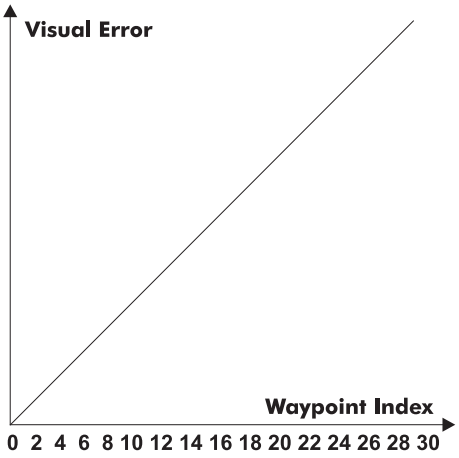
A premise of our error metric is that *any* motion of a visible object that is not received by the client must lead to an *increase* of the visual error, because the user misses information about the behavior of the object that it should perceive.

Hence we *accumulate* the visual error, instead of measuring an absolute value. By adding the relative displacement after each simulation step to the actual error, every motion is accounted for in the error determination. The visual error accumulated in the meantime is set to zero whenever an update of the most recent position is received by the client. Figure 9.2(b) shows how the visual error of the motion depicted in Figure 9.2(a) evolves if the incremental metric is used.

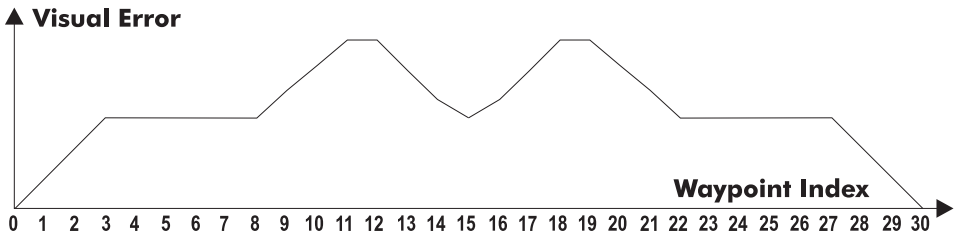
Following this argumentation line, if an object becomes invisible we '*preserve*' the visual error accumulated up to that moment. As long as the object is invisible, it does not accumulate further error, because the *actual* motion of the object cannot be perceived by the user. But the error that was accumulated *before* the object became invisible is still considered in the overall visual error, because even if the object cannot be perceived right now, the user still missed part of the visible motion



(a) Waypoints of the avatar's movement.



(b) Visual error determined incrementally.



(c) Visual error of Figure 9.2(a) as determined by the absolute Euclidian distance.

Figure 9.2: Measurement of the visual error (Euclidian distance).

that it should have perceived. The object continues to accumulate visual error as soon as it becomes visible again.

For example, if an object disappears behind a wall as depicted in Figure 9.3, the contribution of the object to the overall visual error does not increase until it reappears again on the other side of the wall. But the error that was accumulated while heading toward the wall must be preserved. The user missed some information about the object's behaviour, and might still assume the object is located somewhere in front of the wall. Figure 9.4 shows how the visual error accumulates over time for the object in Figure 9.3.

However, the fact that RR and PRR transmit only updates for objects which are visible penalizes the invisible ones. They are forced to 'keep' their visual error (accumulated when they were visible), but are not allowed to send an update to the client, in order to set their accumulated error to zero. Therefore, we allow PRR and RR to update even invisible objects, but only if they were still *visible* the *last time* they were selected (and thus an update was transmitted to the client). In

this way, the user is able to perceive that the object disappeared, and the object is allowed to clear its error contribution. Until the object becomes visible again, any subsequent selections will be ignored by PRR and RR (if the calculation of the TBV was correct, the PRR should not reschedule the element until it gets visible again anyway).

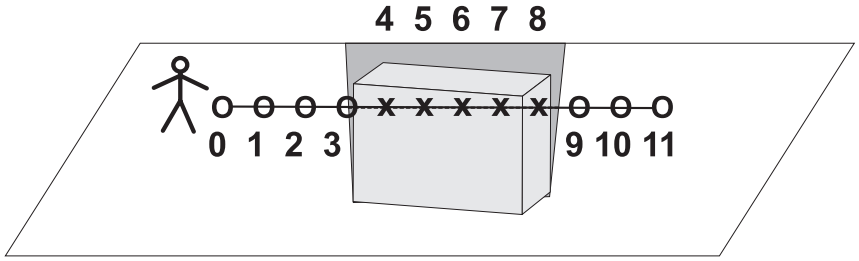


Figure 9.3: Movement of the avatar through an invisible area.

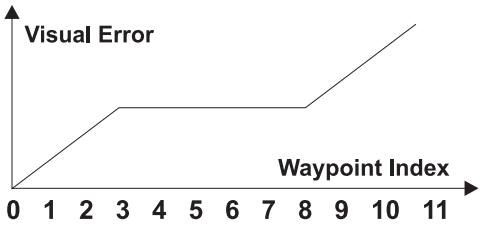


Figure 9.4: Evaluation of the visual error for the movement in Figure 9.3.

Chapter 10

Results

To compare the performance of PRR under various circumstances, we evaluate a given set of error distributions (presented below) on four different setups with increasing complexity:

- **No visibility**: in the first part of the evaluation we do not employ visibility information; we assume that each client can perceive all objects present in the environment (hence all objects selected by the PRR and RR scheduler are effectively transmitted to the client). The scheduling of the objects is based entirely on their velocity with which they are moved along randomized paths. This makes it possible to observe the influence of the error distribution on the performance of the PRR algorithm directly. As there are no client-specific differences between the various clients connected (such as independent viewpoints), the sequence of updates transmitted by PRR is the same for all clients (each clients has its own dedicated PRR scheduler). The RR queues always transmit the same sequence for every client.

- **Visibility**: the second part of the evaluation includes visibility information, given by the position, direction and field of view of a camera that is moved by the client through the environment. As the cameras controlled by the various clients are independent from each other, the priorities as determined by the corresponding PRR queues, and hence the overall errors experienced by the different clients may vary substantially. In most examples we employ a camera with a field of view of 90 degrees that is moved along a randomized path, using the same behavioral rules as for the objects; the velocity of the camera is set to 0.1 units. Figure 10.1 shows the triangulated floorplan employed in our evaluations; for the sake of clarity only 100 objects are displayed. The red star depicts a camera, with the invisible triangles shaded dark. For efficiency reasons we employ a very simple visibility algorithm (see [Schm97]), which does not compute the visible and invisible portions of a triangle. But as the Temporal Bounding Volumes, used to calculate the priorities of the invisible objects, are a very conservative estimate, this does not affect the performance of the PRR algorithm.

- **Visibility and hotspots**: in order to increase the realism of the objects' behavior, we set five sparse 'hotspot' areas in the environment and require that all objects select their destinations to be located in one of the hotspots. This resembles the fact that in virtual environments (especially in online games) there usually exist determined areas of interest such as rooms, buildings, etc., while large portions of the environment are only traversed when migrating from one such area to another. In this setup, every time a client reaches the end of his motion path, with a 75% probability it selects a new destination in a different hotspot; with the remaining 25% the new destination lies in the same hotspot the client is actually located in. In the examples where we employed static cameras, we positioned them near one of the hotspots; when dynamic cameras were used, their motion through the environment was independent from the hotspots, in order to include the case of exploring a sparsely populated part of the environment. Figure 10.2 shows the same floorplan as in Figure 10.1, but with hotspot areas represented by the colored triangles.

- **Visibility, hotspots and leaders**: another typical behavior in virtual environments is the formation of crowds, a fact supported by the existence of the hotspots. We simulate this behavior by designating some objects as 'leaders', which move from on hotspot to another, or to a different location in the same hotspot. The remaining objects then follow one of these leaders (at their own speed).

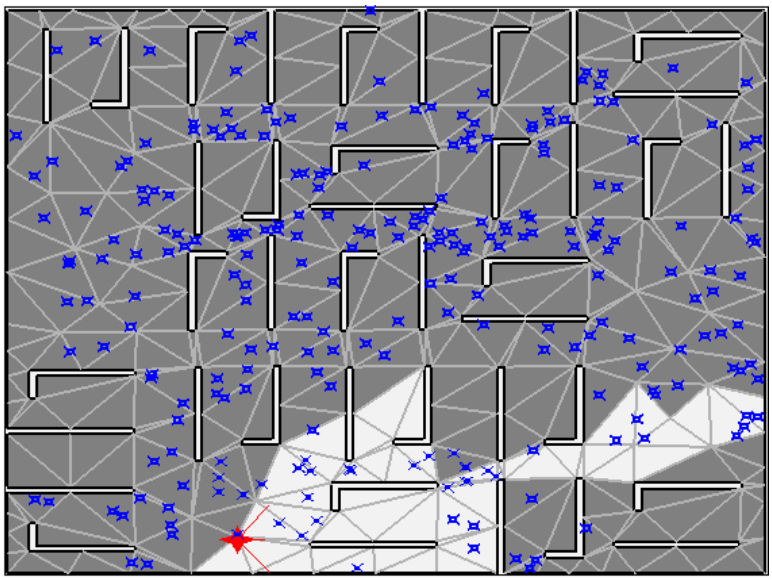


Figure 10.1: Visible area for the camera (star). Invisible triangles are shaded gray.

For the evaluations we populate the environment with 1000 objects. First we assign them a uniform error distribution: each object gets a velocity between 1 and 3 units. Then the analysis is repeated using a clustered error distribution, a case more apt for the PRR algorithm because it allows to construct clearly distinct error groups. We employ two different sets, denominated $'S'$ (for 'small') and $'L'$ (for 'large'): they resemble the fact that in many virtual environments a small set of objects has a high velocity with respect to the majority of the objects; for example, if humans, vehicles or even airplanes are simulated contemporaneously in the same environment. Basically, the more distinct the groups are, the higher the performance increase achieved by PRR.
The first set, called $'S'$, has three different error groups:

- 100 objects have a velocity between 9 and 10 units.
- 250 objects have a velocity between 1 and 1.2 units.
- 650 objects have a velocity between 0.1 and 0.5 units.

In the second set $'L'$ the groups are more distinct:

- 100 objects have a velocity between 9 and 10 units.
- 250 objects have a velocity between 0.1 and 0.12 units.
- 650 objects have a velocity between 0.01 and 0.05 units.

The evaluation of these three error distributions in the different environment setups is presented in Sections 10.1 to 10.4. To show how the RR and PRR algorithms react to a varying network bandwidth, we repeat all examples with two different settings, allowing the schedulers to update 50 or 100 objects after each simulation loop, respectively.

All evaluations are performed with 10 clients connected simultaneously to the server, each moving a dedicated camera through the environment (except in Sections 10.1 where visibility is ignored). The graphs in the following sections show the overall visual error for both PRR and RR (y-axis), which has been summed up over all ten clients; the x-axis of the graphs is labelled with the simulation loops (or main loops) performed.

The performance increase achieved by PRR is expressed as percentage value. Using the error of RR as reference value (100%), we determine by how many percent the *overall visual error* can be reduced if PRR is used instead of RR. A result of 42%, for example, means that the visual error of PRR is 58% the error of RR. The percentage given refers to the sum of the individual errors of all client; as there may be substantial variations, we also list the percentages of the single clients.

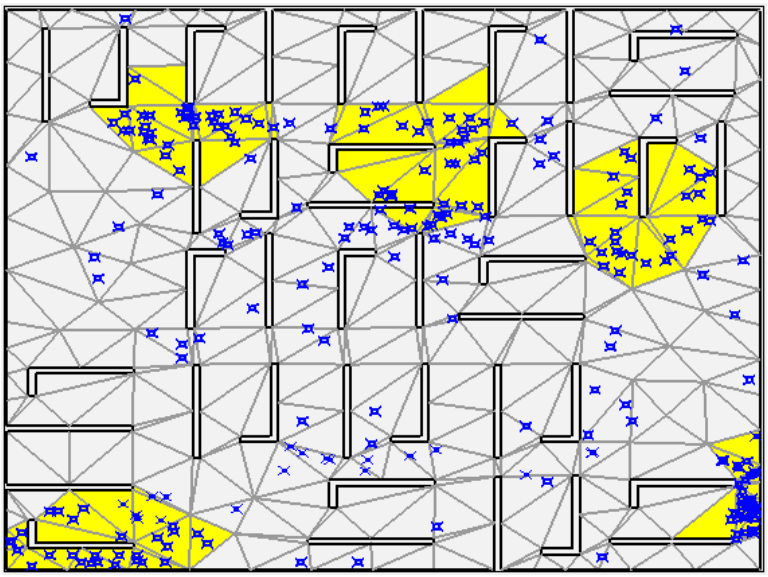


Figure 10.2: Environments with five hotspot-areas (no camera drawn).

Figure 10.3 presents a snapshot of the testbed implementation. The central panel shows the triangulated floorplan with the objects simulated by the server (for the sake of clarity only 100 objects are drawn). The left and right panels on top of the snapshot visualize the individual errors of the single objects as experienced by one specific client; the left panel shows the visual error for the RR queue, while the right panel is associated to the PRR scheduler. It is possible to select between different types of visualization: the magnitude of the visual error can be represented e.g. by the radius of a circle centered around the objects (as shown in Figure 10.3), by a line connecting the position of the object on the server with the last position update received by the client, or varying the color brightness with which the object is drawn. The graph at the bottom of the snapshot displays the *overall visual error* for the PRR and the RR algorithm (summed up over all clients).

A fact that can be observed in all examples is that by employing a uniform error distribution, the performance increase that can be achieved with the PRR algorithm is much more limited with respect a 'clustered' error distribution, because the algorithm cannot construct clearly distinct error groups (to be serviced at different priorities).

Furthermore, when employing a uniform error distribution, the size of the interval has only a restricted impact on the performance of PRR, as can be seen in Sections 10.1: the intervals ranging from [0.1, 1] up to [1, 200] show almost the same results (a decrease of the visual error by 10% on average). But by using clustered error distributions, we measured performance increases of up to 7600(!) percent (shown in Graph 10.22). If visibility information is employed, the size of the uniform error interval has a larger impact on the PRR performance, mainly due to the use of Temporal Bounding Volumes.

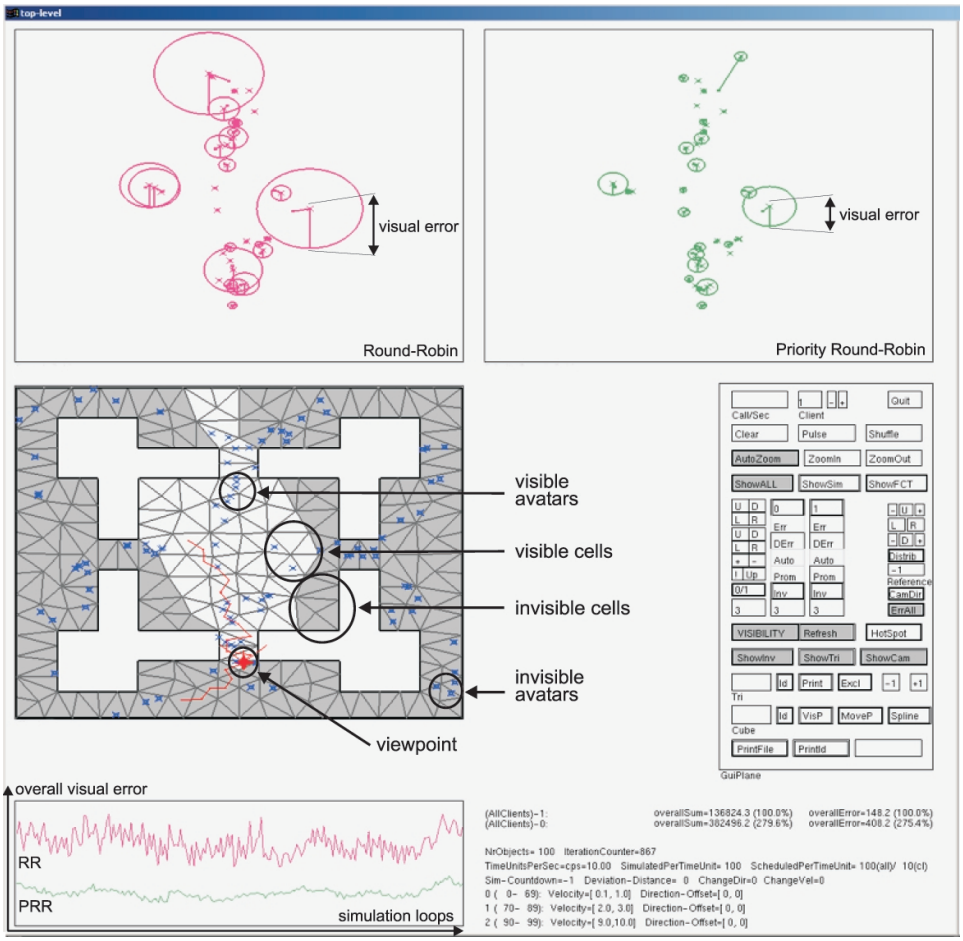


Figure 10.3: Snapshot of the testbed employed to evaluate the PRR algorithm.

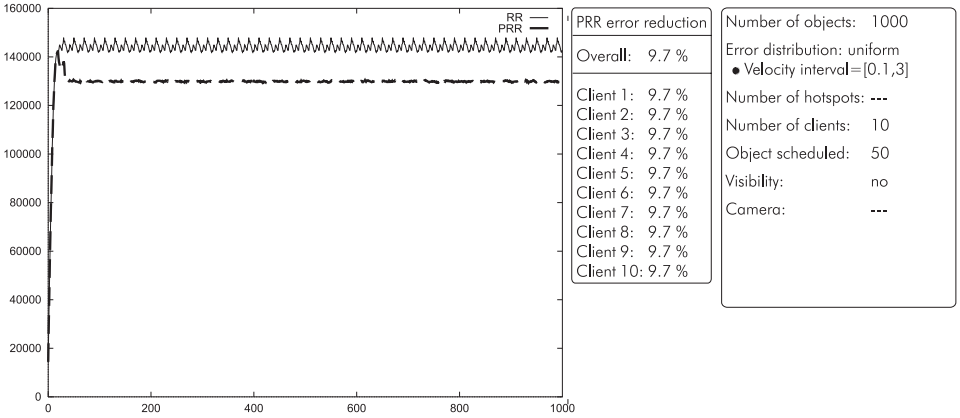
We can also observe that the more objects can be scheduled on each loop, the bigger is the advantage of PRR over RR scheduling. With uniform error distributions, the difference between scheduling 50 or 100 objects on each loop is minimal, but with clustered error distributions there may be substantial differences.

10.1 No visibility

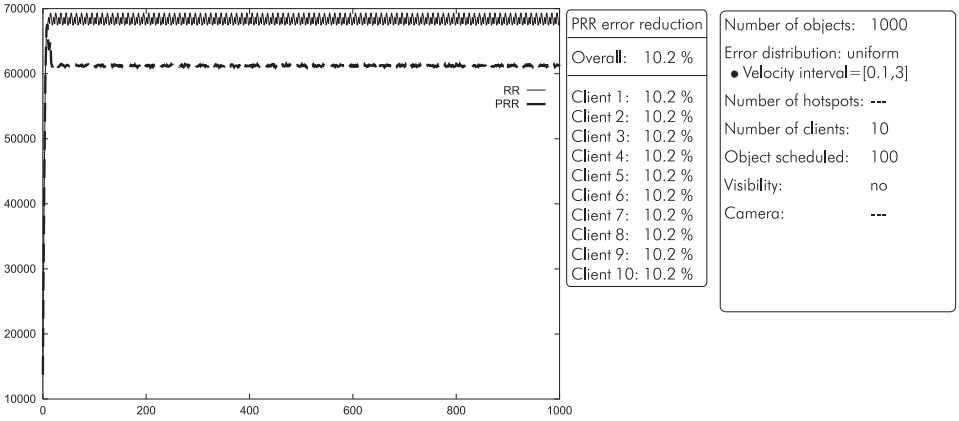
First we evaluate the performance of the PRR algorithm without employing visibility information, thus no camera is associated to the clients.

We rather assume that all objects are visible to each client. Graph 10.1 shows the overall visual error of PRR and RR if all 1000 objects is assigned a velocity between 0.1 and 3 units; both algorithms are allowed to schedule 50 objects (5% of all objects) after each simulation loop. In this case, the overall visual error of PRR is 9.7% lower than that generated by RR.

Graph 10.2 shows the same configuration, but this time allowing PRR and RR to schedule 100 objects (10% of all objects). The magnitude of the visual error (y-axis) is clearly much lower than compared to scheduling only 50 objects; the relation between both errors (RR and PRR) changes only slightly: the reduction of the overall error by employing PRR is 10.2%.



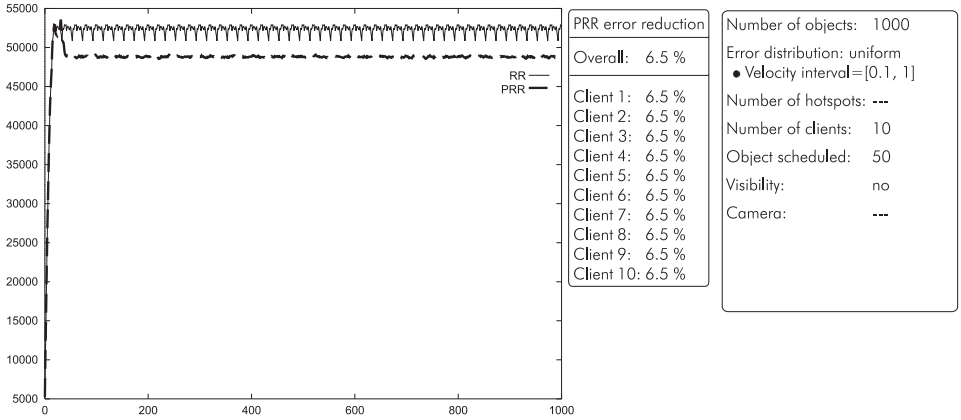
Graph 10.1: The overall visual error of RR is reduced by 9.7% using PRR scheduling.



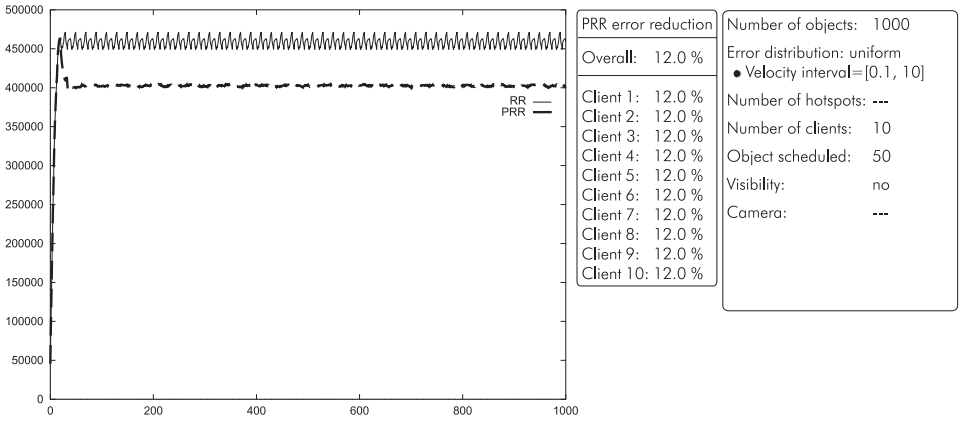
Graph 10.2: The overall visual error of RR is reduced by 10.2% using PRR scheduling.

In both graphs it is possible to observe the frequency with which the plain RR queue traverses the objects: the period of the RR error curve is exactly 20 loops (hence, 20 simulation steps) in Graph 10.1, and 10 steps in Graph 10.2. The reason is that RR requires exactly 20 loops to traverse the whole queue once if 50 objects are scheduled per loop; with 100 scheduled objects the turnaround time is lowered to 10 loops. This effect can be perceived even more clearly in the examples which employ clustered error distributions. In all cases the PRR algorithm is not only able to achieve a lower overall error than RR, but it also keeps the visual error mostly constant, while the visual error of the RR queue can have very large amplitudes (see e.g. Graphs 10.5 through 10.7).

To show that by employing a uniform error distribution the size of the interval has only a limited effect on the performance of PRR, Graph 10.3 depicts the visual error if all objects is assigned a velocity between 0.1 and 1 units, while Graph 10.4 visualizes a velocity between 0.1 and 10 units (scheduling 50 objects). In the first case we can lower the visual error by 6.5% if we employ PRR instead of RR; in the second case the reduction is exactly 12%.



Graph 10.3: The overall visual error of RR is reduced by 6.5% using PRR scheduling.

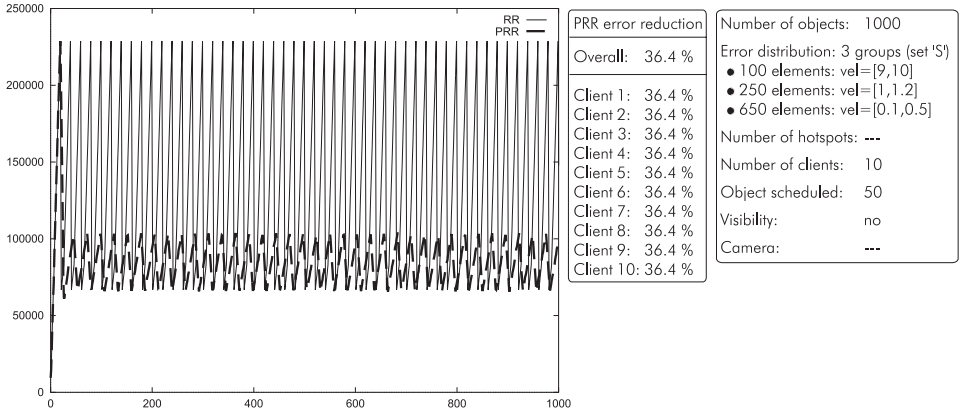


Graph 10.4: The overall visual error of RR is reduced by 12% using PRR scheduling.

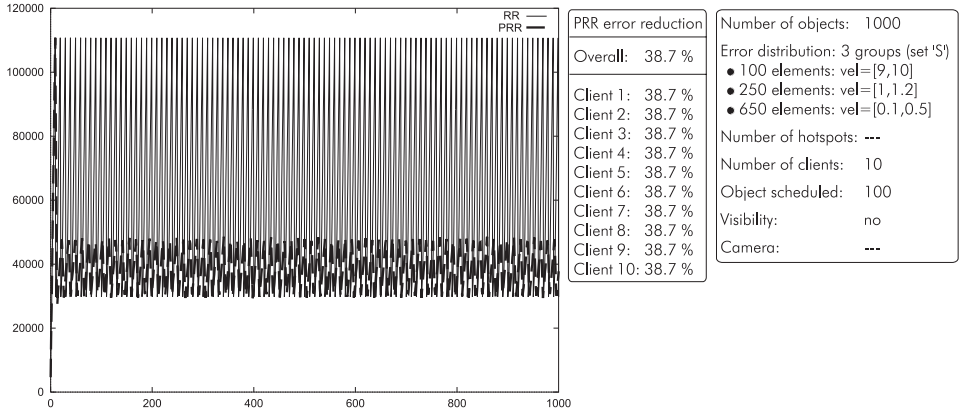
Even if we increase the uniform error interval by a larger amount, the performance changes marginally. By assigning all objects a velocity between 0.1 and 100 units, the reduction of the visual error is 11.2% when scheduling 50 objects and 11.7% when scheduling 100 objects per loop. With an error interval between 0.1 and 200 the overall error produced by PRR is 10.7% and 11.2% lower than RR(for 50 and 100 object per loop, respectively).

If we employ a clustered error distribution, the PRR algorithm is able to construct clearly distinct groups handled at different priorities. By employing the clustered set $'S'$ (3 groups: 100 objects have a velocity between 9 and 10 units, 250

objects have a velocity between 1 and 1.2 units, and the remaining 650 objects between 0.1 and 0.5 units), the PRR algorithm lowers the overall visual error by 36.4% if 50 objects are scheduled after each simulation step (Graph 10.5), and 38.7% for 100 updated objects (Graph 10.6).

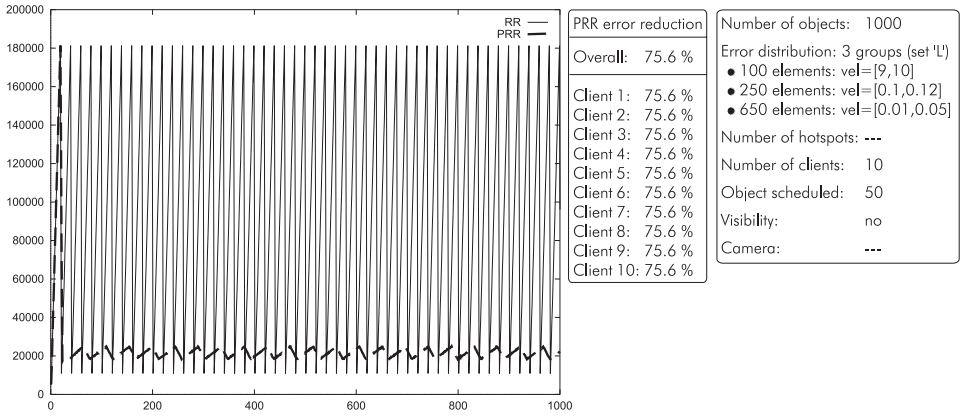


Graph 10.5: The overall visual error of RR is reduced by 36.4% using PRR scheduling.



Graph 10.6: The overall visual error of RR is reduced by 38.7% using PRR scheduling.

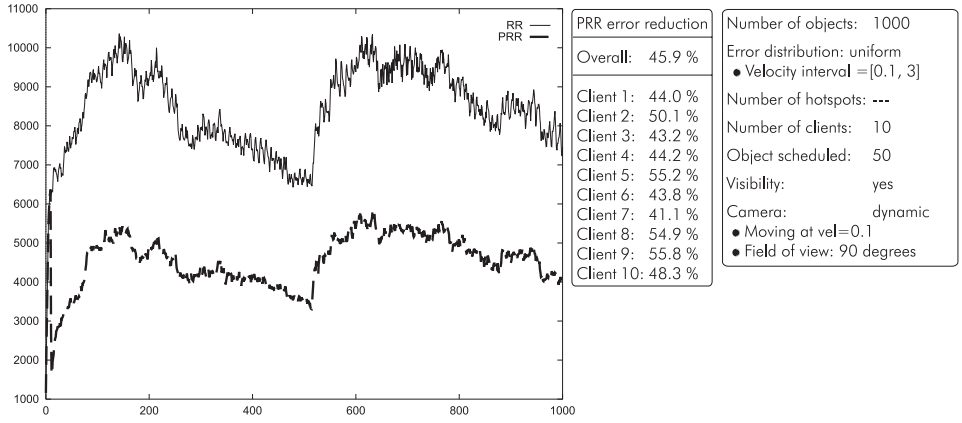
With the second clustered set that has even more distinct error distributions (100 objects: velocity between 9 and 10 units, 250 objects: velocity between 1 and 1.2 units, 650 objects: velocity between 0.1 and 0.5 units) the PRR algorithm can achieve an overall visual error that is 75.6% lower than that of a plain RR queue (for 50 objects scheduled after each simulation step, in Graph 10.7). If 100 objects are scheduled on each loop, the error reduction is 79.6%.



Graph 10.7: The overall visual error of RR is reduced by 75.6% using PRR scheduling.

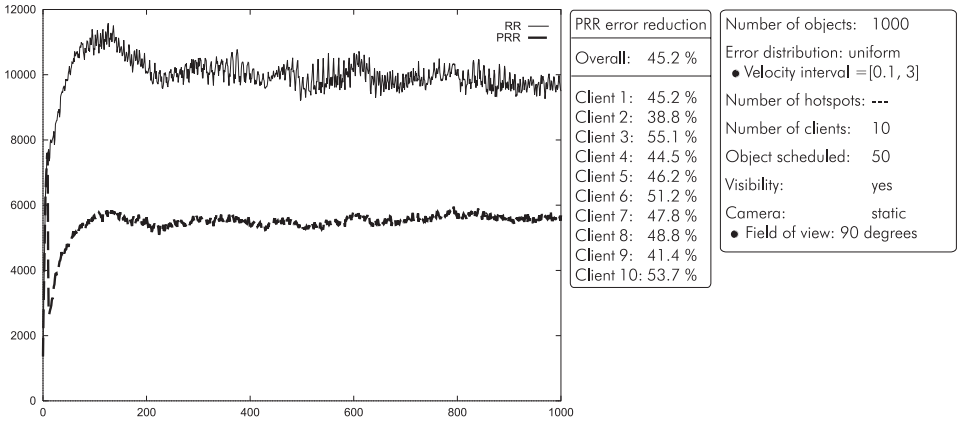
10.2 Visibility

In the following three subsections we employ visibility information in the PRR algorithm when determining the priorities of the objects. The visibility for each client is determined by the position, direction and field of view of an independent camera that is moved through the environment along a randomized path with a velocity of 0.1 units; the field of view is set to 90 degrees. As opposed to Sections 10.1, the reduction of the visual error achieved by PRR for the different clients can vary perceptibly, because the path of the camera and hence the set of objects in its viewcone is different. When using clustered error distributions, these differences are more pronounced than with a uniform error distribution.



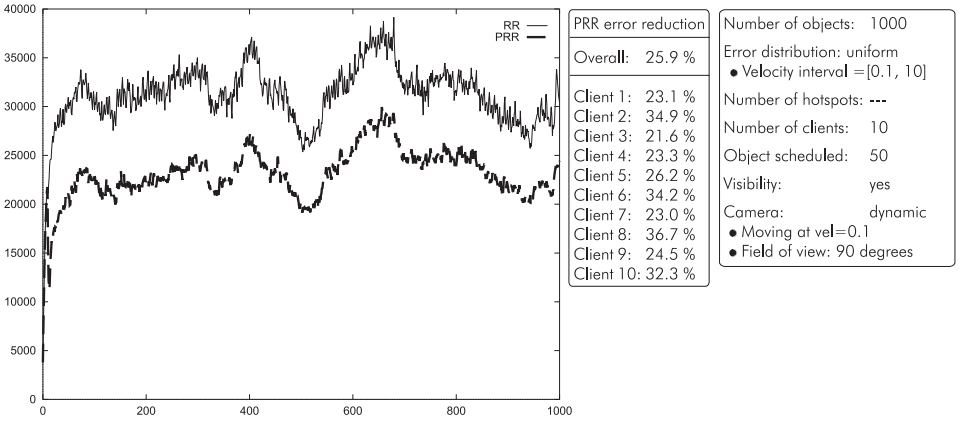
Graph 10.8: The overall visual error of RR is reduced by 45.9% using PRR scheduling.

The first example employing visibility is performed with the uniform error distribution (velocity between 0.1 and 3 units). Graph 10.8 shows the overall visual error for PRR and RR when scheduling 50 objects; the visual error is reduced by 45.9%. If 100 objects are updated per simulation loop, the reduction of the visual error is 52.8%. The steep slope of the curve that can be observed in the middle of the graph was caused by some of the cameras suddenly appearing behind a wall, experiencing an abrupt increase of objects in their viewcone. In the error measurement we consider only the visible objects (please refer to Sections 9.3); this leads to a sudden increase in the number of objects contributing to the visual error.



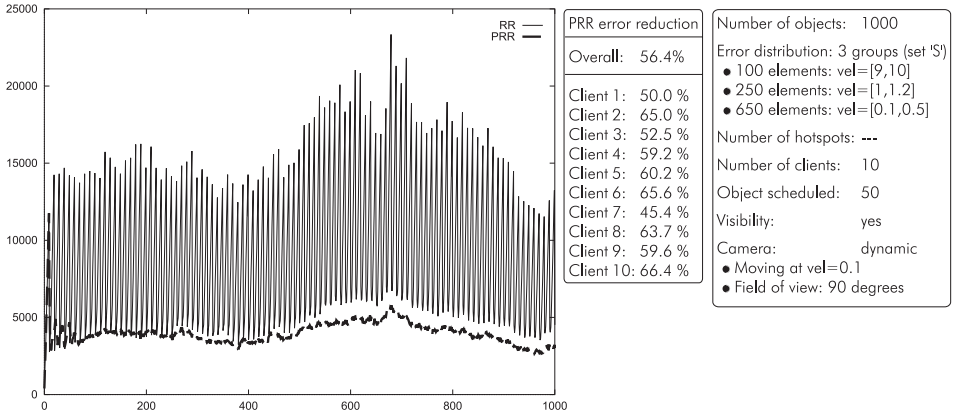
Graph 10.9: The overall visual error of RR is reduced by 45.2% using PRR scheduling.

Graph 10.9 shows the same example, this time with static camera placed at random positions and facing approximately the center of the environment; the reduction of the overall visual error is similar to the evaluation in Graph 10.8: 45.2%. These examples show that the PRR algorithm can cope efficiently with rapidly changing visibility situations.



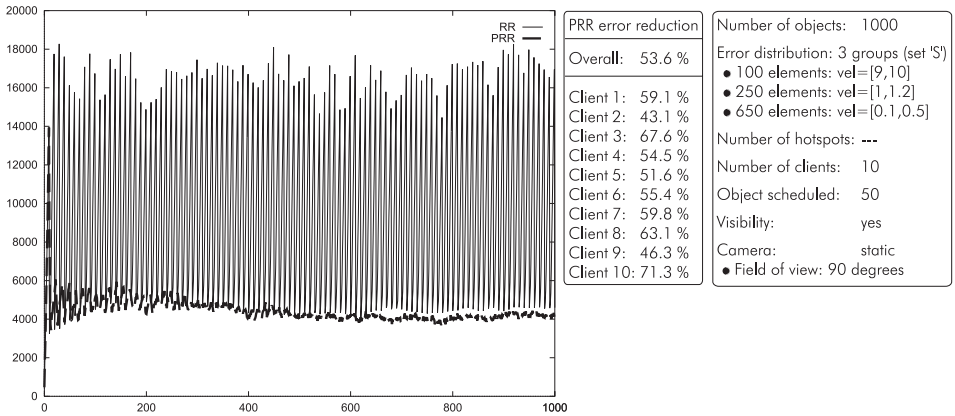
Graph 10.10: The overall visual error of RR is reduced by 25.9% using PRR scheduling.

If we employ a uniform error distribution with a larger interval (e.g. between 0.1 and 10 units), the advantage achieved by PRR over RR is slightly less than for the example in Graph 10.8. The reason lies mainly in the fact that with fast moving objects frequently entering and exiting the viewcones of the cameras, the PRR can perform a less accurate prediction for the object's behavior (the epu for visible and invisible objects is computed in a different way; refer to Chapter 7). Graph 10.10 shows the overall errors for the error interval $\in$ [0.1, 10] with 50 objects scheduled per simulation loop (dynamic camera); the reduction is 25.9%. Scheduling 100 objects lowers the overall visual error by 41.1%.



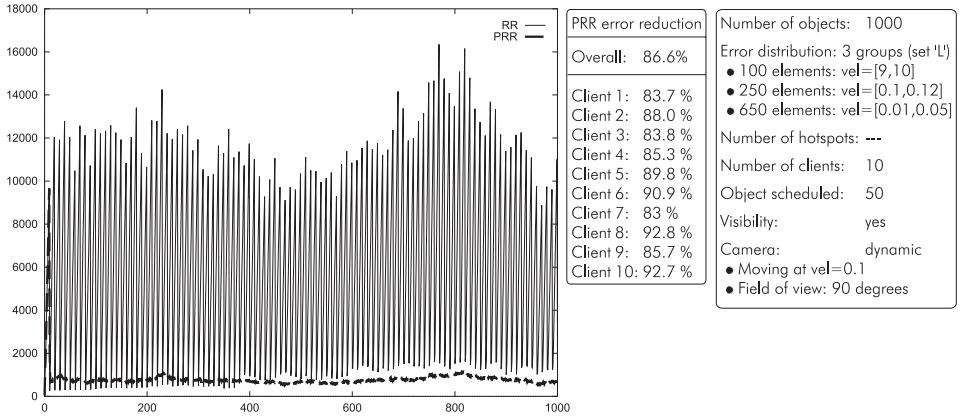
Graph 10.11: The overall visual error of RR is reduced by 56.4% using PRR scheduling.

However, for clustered error distributions the availability of clearly distinct error groups outweighs the reduced predictability caused by changing visibility situations by far. Assigning 100 objects a velocity between 9 and 10 units, 250 objects a velocity between 1 and 1.2 units, and 650 objects a velocity between 0.1 and 0.5 units (set $'S'$), the visual error of PRR is 56.4% lower than that of RR (scheduling 50 objects per simulation loop, Graph 10.11). If 100 objects are scheduled, the advantage is 69.1%.



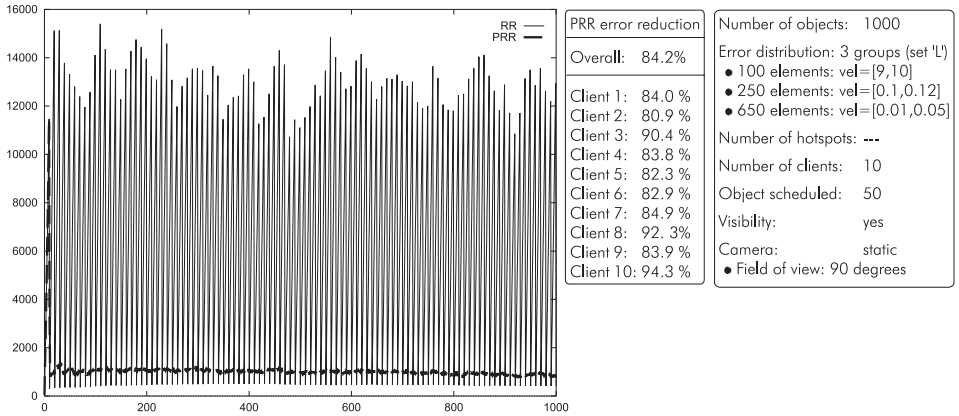
Graph 10.12: The overall visual error of RR is reduced by 53.6% using PRR scheduling.

The motion of the camera has also a slight effect on the evaluation of the overall visual error; Graph 10.12 shows the example of Graph 10.11 repeated with a static camera (random position, direction approximately toward the center of the environment). The reduction of the overall visual error achieved by PRR is almost the same as in Graph 10.11: if 50 objects are scheduled, 53.6%, and if 100 objects are scheduled, 71.8%.



Graph 10.13: The overall visual error of RR is reduced by 86.6% using PRR scheduling.

If we employ set $'L'$ (100 objects: velocity between 9 and 10, 250 objects: velocity between 0.1 and 0.12, 650 objects: velocity between 0.01 and 0.05) together with the dynamic cameras, the overall error is reduced by 86.6% when scheduling 50 objects, as illustrated in Graph 10.13, and by 95.5% when scheduling 100 objects.



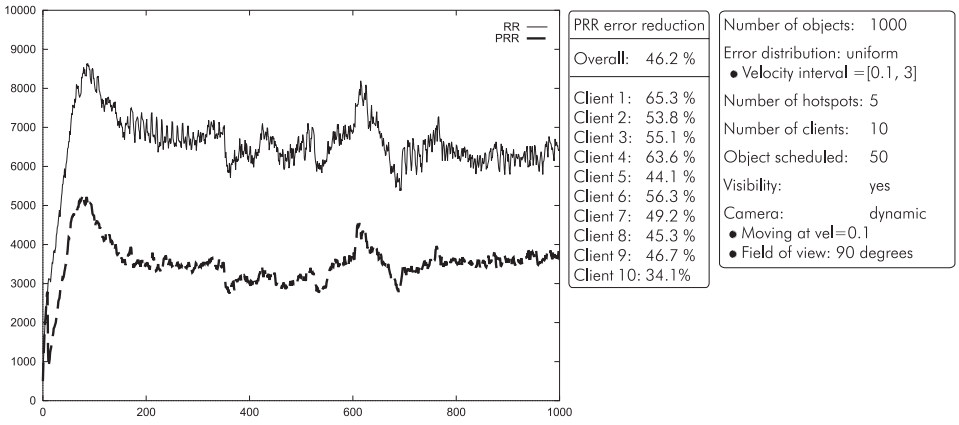
Graph 10.14: The overall visual error of RR is reduced by 84.2% using PRR scheduling.

The more distinct the error groups, the more the graphs for the dynamic and static cameras look similar to each other; Graph 10.14, which repeats the example of Graph 10.13 with a static camera, differs only slightly from Graph 10.13. With a static camera, scheduling 50 objects in each loop allows PRR to reduce the overall visual error by 84.2%; with 100 objects selected per loop the error reduction of PRR is 95.2%.

10.3 Visibility and hotspots

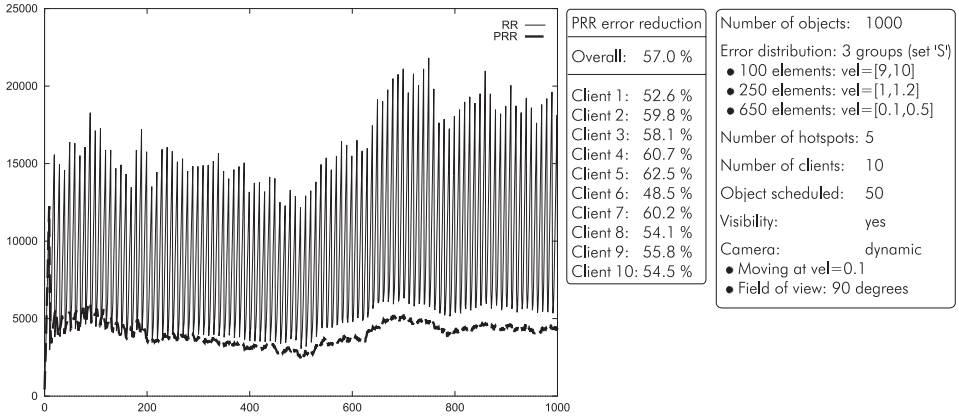
To increase the realism of the objects' behavior, we randomly select five sparse areas in the environment, so called 'hotspots', and employ them as preferred haunts for the objects: we require that all objects select their destinations to be located in one of the hotspots. This resembles the fact that in large virtual environments, and especially in online games, the avatars are usually interested in few dedicated areas of the environment, such as rooms, buildings, cities etc., while most parts of the environment are only traversed when migrating from one such area to another. Every time a client reaches the end of a previously selected path, it chooses a new destination; with a probability of 75% the new destination will lie in a different hotspot, with the remaining 25% it lies on a different position of the same hotspot the object is actually located in. The path of the cameras is not affected by the hotspots, for not sticking to the hotspots and include the case of exploring the sparsely populated parts of the environment.

The evaluation for the uniform error distribution (velocity interval between 0.1 and 3 units) is presented in Graph 10.15. By scheduling 50 objects, the overall visual error of PRR is 46.2% lower compared to RR; updating 100 objects in each loop, the overall error can be decreased by 52.0%.



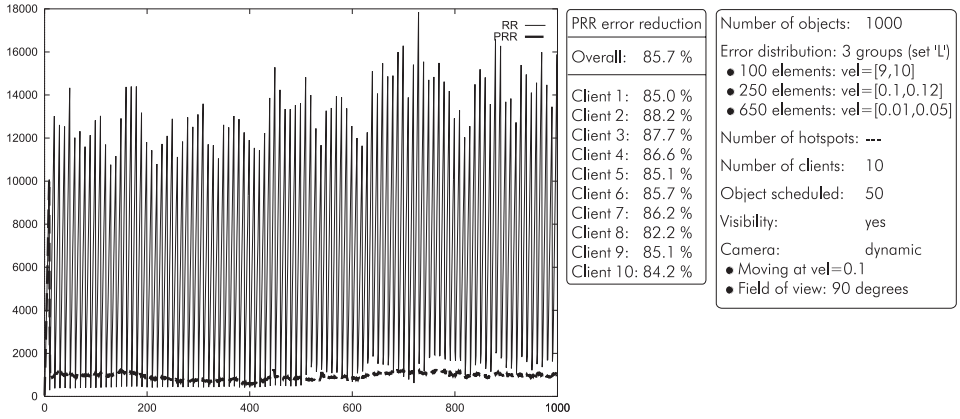
Graph 10.15: The overall visual error of RR is reduced by 46.2% using PRR scheduling.

With the clustered error set $'S'$, the performance of PRR is as follows: if 50 objects are scheduled in each loop, the overall visual error is reduced by 57% (Graph 10.16); with 100 objects scheduled the reduction is 70%.



Graph 10.16: The overall visual error of RR is reduced by 57.0% using PRR scheduling.

The clustered error set $'L'$ (100 objects $\in$ [9,10], 250 objects $\in$ [0.1, 0.12], 650 objects $\in$ [0.01,0.05]) shows a reduction of the visual error by 85.7% according to Graph 10.17 (scheduling 50 objects). If 100 objects are scheduled, PRR lowers the error by 95.4%.



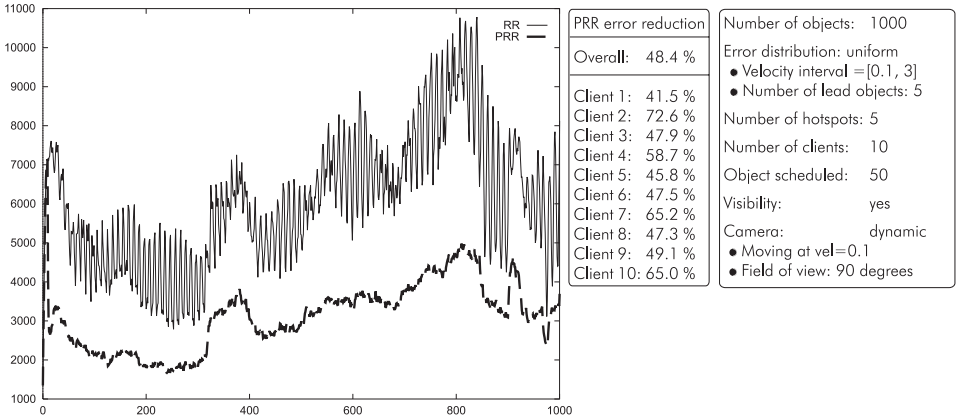
Graph 10.17: The overall visual error of RR is reduced by 85.7% using PRR scheduling.

10.4 Visibility, hotspots and leaders

In this section we enhance the simulation of the objects by another behavior that can often be observed in large scale virtual environments and online games, namely the formation of crowds; gathering together and moving in groups is an attitude that is partly encouraged by the existence of the haunt areas (hotspots). We designate some objects as 'leaders'; they repeatedly move from one hotspot to another (with a probability of 75%) or to a different position in the same hotspot (with a probability of 25%), at the speed specified by the error distribution. The objects which are not 'leaders' are designated as 'followers': they randomly select one of the leader-objects and try to follow its movements (they always select the same destination as their leaders do). Because the follower-objects move at their own speed, whenever clustered error distributions are employed, the followers always choose a leader from the same error group, otherwise the velocity difference might by too high to follow its movements).

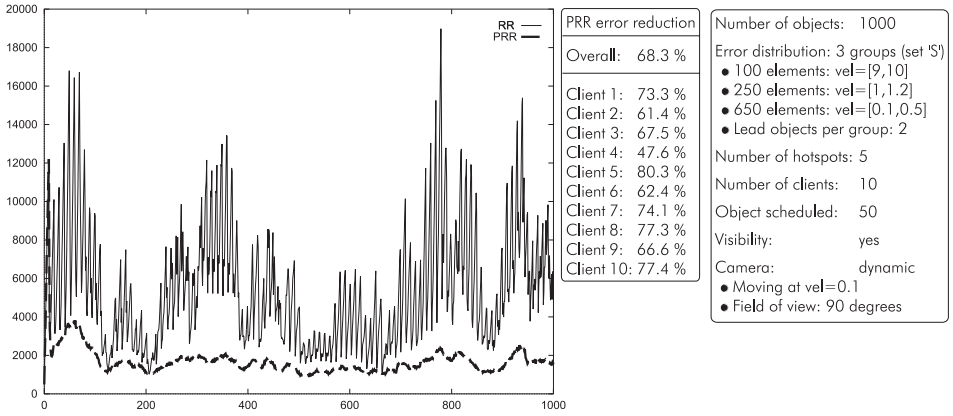
The first example, shown in Graph 10.18, is performed with the uniform error distribution $\in$ [0.1,3]; we choose 5 of the 1000 objects as leaders. The remaining objects select one of these five objects, and follow every its movement.

We can observe five different groups of objects (of approximately equal size) following their leader as it moves from hotspot to hotspot. The cameras are dynamic (velocity = 0.1 units); their path is not affected by the hotspots. If 50 objects are scheduled after each simulation step, the overall visual error of PRR is 48.4% lower compared to RR; when scheduling 100 objects, PRR reduces the visual error by 53.8%.



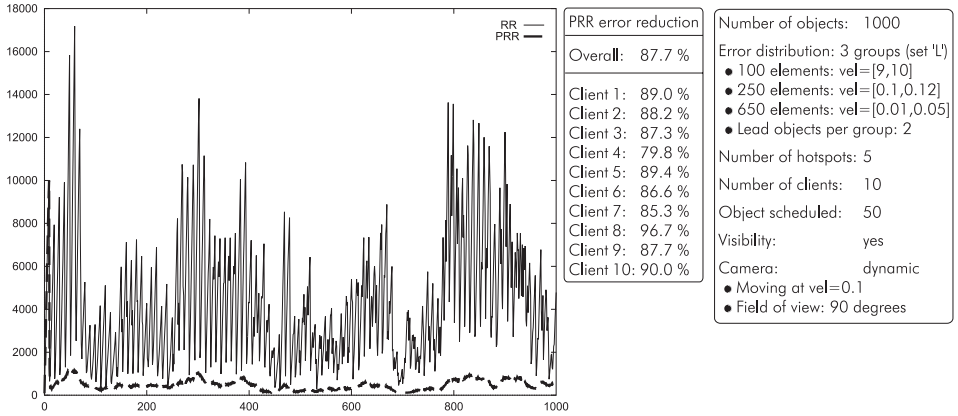
Graph 10.18: The overall visual error of RR is reduced by 48.4% using PRR scheduling.

We repeat the example with the clustered error set $'S'$ (Graph 10.19). This time we designate two leaders per group, and let the remaining objects in each group follow one of both 'alpha' objects. Therefore we have 6 different groups moving through the environment, in three different speed clusters. The behavior of the cameras is the same as in the previous example. PRR reduces the overall visual error of RR by 68.3% (50 objects scheduled per loop) and 78.1% (100 objects scheduled per loop).



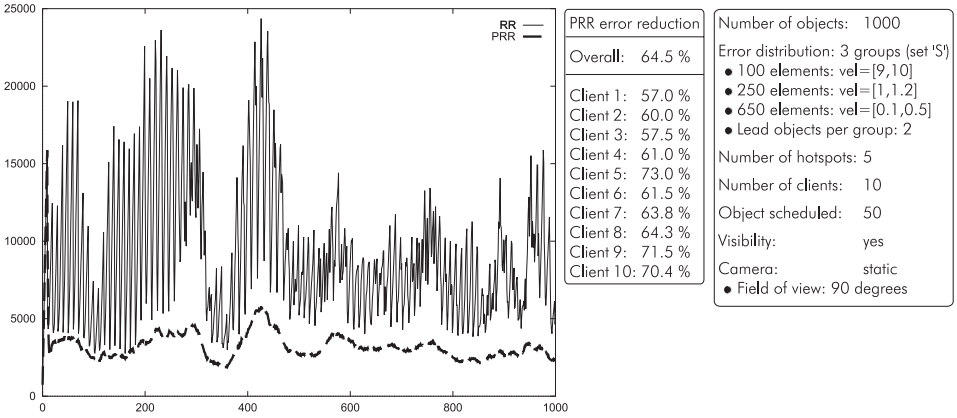
Graph 10.19: The overall visual error of RR is reduced by 68.3% using PRR scheduling.

With the clustered error set $'L'$ (100 objects $\in$ [9, 10], 250 objects $\in$ [0.1, 0.12] and 650 objects $\in$ [0.01, 0.05]), the visual error of PRR is 87.7% lower compared to RR (scheduling 50 objects, Graph 10.20), and 96.4% lower when scheduling 100 objects per loop. Again two objects were designated as leaders in each of the three error groups.



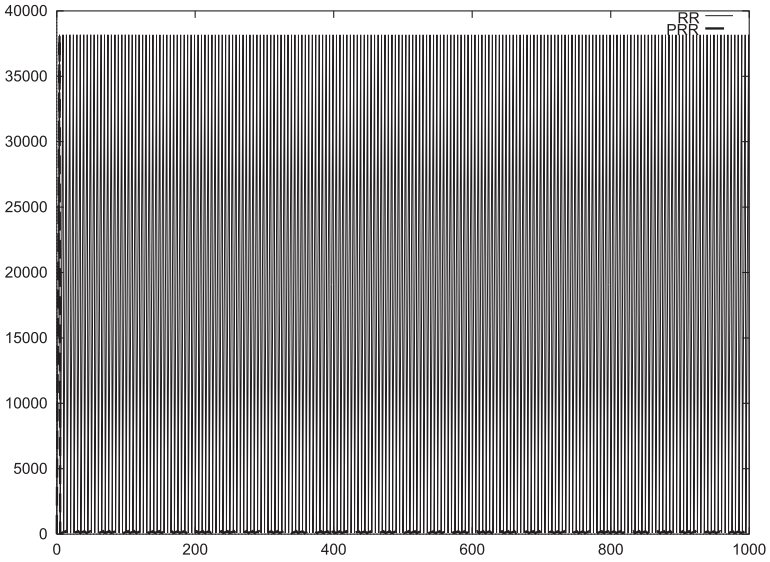
Graph 10.20: The overall visual error of RR is reduced by 87.7% using PRR scheduling.

For the sake of completeness we repeat the example of Graph 10.19 with static cameras; to avoid letting them face an area empty most of the time, we position them near one of the hotspots. The result is shown in Graph 10.21, with a reduction of the overall error by 64.5% (50 objects scheduled per loop). If 100 objects are scheduled, the visual error of PRR is 72.5% lower than that of RR.



Graph 10.21: The overall visual error of RR is reduced by 64.5% using PRR scheduling.

Concluding, a rather theoretical example, but that clearly shows the potential of the Priority Round-Robin algorithm. We assume the situation of 10% of the objects having a velocity 1000 times higher than the remaining objects; if airplanes are simulated contemporaneously with pedestrians, we might have a situation like this. 100 objects have a velocity between 9 and 10 units, 250 objects have a velocity between 0.01 and 0.012 units, and the 650 remaining objects between 0.001 and 0.005 units. By scheduling 200 objects on each loop, the visual error of PRR is 98.7% lower than RR - this means the visual error of RR is 7508% higher than that achieved by using PRR, and whose curve has quite an elevated amplitude, as can be seen from Graph 10.22 (the PRR curve can be hardly detected at the very bottom of the graph).



PRR error reduction	
Overall:	98.7 %
Client 1:	98.7 %
Client 2:	98.7 %
Client 3:	98.7 %
Client 4:	98.7 %
Client 5:	98.7 %
Client 6:	98.7 %
Client 7:	98.7 %
Client 8:	98.7 %
Client 9:	98.7 %
Client 10:	98.7 %

Number of objects:	1000
Error distribution:	3 groups
	• 100 elements: vel=[9,10]
	• 250 elements: vel=[0.01,0.012]
	• 650 elements: vel=[0.001,0.005]
Number of hotspots:	---
Number of clients:	10
Object scheduled:	200
Visibility:	no
Camera:	---

Graph 10.22: The overall visual error of RR is reduced by 98.7% using PRR scheduling.

Chapter 11

Conclusion

We have presented a technique to enhance plain Round-Robin scheduling by adding the enforcement of priorities to its advantages of being output sensitive and immune to starvation. The Priority Round-Robin (PRR) algorithm can bring a substantial contribution to the development of distributed virtual environments and networked online-games containing a very high number of objects. The simplicity of PRR and the freely definable error metric make it a suitable substitution for Round-Robin in most cases. In our examples we have employed PRR as substitute for the plain Round-Robin queue to transmit the update messages at a constant effort per connected client; the frequency of the updates is determined from priorities based on the behavior of the objects. Furthermore, PRR can be efficiently combined with reduction/filtering techniques such as visibility culling. By including the visibility information in the determination of the objects' priorities, we do not abandon output sensitivity. Hence PRR not only provides a scalable techniques that leads to a graceful degradation of the system's performance concerning network bandwidth limitations. But it also helps avoid computational bottlenecks caused by a naive application of reduction/filtering techniques. To account for the often unpredictable or rapidly changing behavior of user controlled avatars, especially in online games, a heuristic measure for the activity of the objects was also incorporated in our system. Whenever the motion the objects is too unpredictable to base the determination of their priorities on the recent simulation behavior, the influence of the priorities is successively restricted, which leads to a graceful degradation of PRR performance in almost any circumstances. Although our evaluations were performed with only 1000 objects and 10 clients connected, the PRR algorithm has no theoretical restrictions, thus being applicable to environments of any size. Therefore, it is a valid contribution to the construction of scalable large scale distributed virtual environments and networked online games.

Bibliography

[Aire90] J. M. Airey, J. H. Rohlf, and F. P. Brooks. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, Vol. 24(2), pages 41–50, 1990.

[Barr96] J. W. Barrus, R. C. Waters, and D. B. Anderson. Locales and Beacons: Precise and Efficient Support for Large Multi-User Virtual Environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'96)*, pages 204–213, 1996.

[Calv93] J. Calvin, A. Dicken, B. Gaines, P. Metzger, D. Miller, and D. Owen. The SIMNET virtual world architecture. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 450–455, 1993.

[Calv95] J. Calvin, J. Seeger, G. Troxel, and D. Van Hook. STOW real-time information transfer and networking system architecture. In *Proceedings of the 12th DIS Workshop*, 1995.

[Capi97a] T. K. Capin, I. S. Pandzic, N. Magnenat-Thalmann, and D. Thalmann. A Dead-Reckoning Algorithm for Virtual Human Figures. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'97)*, pages 161–168, 1997.

[Capi97b] T. K. Capin, I. S. Pandzic, H. Noser, N. Magnenat-Thalmann, and D. Thalmann. Virtual Human Representation and Communication in VLNET. *IEEE Computer Graphics and Applications*, Vol. 17(2), pages 42–53, 1997.

[Carl93] C. Carlsson and O. Hagsand. DIVE - A multi-user virtual reality system. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 394–400, 1993.

[Carl97] D. A. Carlson and J. K. Hodgins. Simulation levels of detail for real-time animation. *Graphics Interface '97*, pages 1–8, 1997.

[Clar76] J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, Vol. 19(10), pages 547–554, 1976.

[Come94] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume 1*. Prentice Hall, 1994.

[Dans94] J. Danskin. Higher Bandwidth X. *ACM Multimedia'94*, pages 89–96, 1994.

[Das97] T. K. Das, G. Singh, A. Mitchell, P. S. Kumar, and K. McGhee. NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'97)*, pages 157–164, 1997. ISBN 0-89791-953-X.

[Deer89] S. E. Deering. Host extensions for IP multicasting, 1989. Internet RFC 1112, Information Sciences Institute, Marina Del Rey, CA. Available at `http://info.internet.isi.edu/in-notes/rfc/files/rfc1112.txt`.

[Deer95] M. F. Deering. Geometry Compression. *Computer Graphics (SIGGRAPH '95 Proceedings)*, Vol. 29, pages 13–20, 1995.

[Deit90] H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, 1990. ISBN 0-201-18038-3.

[El-R98] H. El-Rewini and T. G. Lewis. *Distributed and Parallel Computing*. Manning Press, 1998.

[Fais00a] C. Faisstnauer, D. Schmalstieg, and W. Purgathofer. Priority Round-Robin Scheduling for Very Large Virtual Environments. In *Proceedings of the 2000 IEEE Conference on Virtual Reality (VR'00)*, pages 135–142, 2000.

[Fais00b] C. Faisstnauer, D. Schmalstieg, and W. Purgathofer. Priority Scheduling for Networked Virtual Environments. *IEEE Computer Graphics and Applications*, Vol. 20(6), pages 66–75, 2000.

[Fais00c] C. Faisstnauer, D. Schmalstieg, and W. Purgathofer. Scheduling for Very Large Virtual Environments and Networked Games Using Visibility and Priorities. In *Proceedings of the DIS-RT 2000 conference*, pages 31–38, 2000.

[Funk95] T. A. Funkhouser. RING: A Client-Server System for Multi-User Virtual Environments. In *Proceedings of the 1995 SIGGRAPH Symposium on Interactive 3D Graphics*, pages 85–92, 1995. ISBN 0-89791-736-7.

[Funk96a] T. A. Funkhouser. Network Topologies for Scalable Multi-User Virtual Environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'96)*, pages 222–229, 1996.

[Funk96b] T. A. Funkhouser, S. Teller, C. Sequin, and D. Khorramabadi. The UA Berkeley System for Interactive Visualization of Large Architectural Models. *Presence (Virtual Reality and Teleoperators)*, Vol. 5(1), pages 13–44, 1996.

[Gree95] C. Greenhalgh and S. Benford. MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 27–35, 1995. ISBN 0-8186-7025-8.

[Gree97] C. Greenhalgh and S. Benford. Boundaries, awareness, and interaction in collaborative virtual environments. In *Proceedings of the Sixth IEEE Workshop on Enabling Technologies (WETICE)*, pages 193–198, 1997.

[Grim91] C. Grimsdale. dVS - Distributed Virtual Environment System. In *Proceedings of the Computer Graphics, Computer Animation, Virtual Reality, Visualization '91 Conference*, 1991. ISBN 0-86353-282-9.

[Harv97] W. Harvey. The future of Internet games. *Modelling and Simulation - Linking Entertainment and Defense. Computer Science and Telecommunications Board*, pages 140–143, 1997.

[Heck97] P. S. Heckbert and M. Garland. Survey of Polygonal Surface Simplification Algorithms. Technical Report, CS Department, Carnegie Mellon University, 1997. Available at `http://www.cs.cmu.edu/~ph`.

[httpDIVE] DIVE web site: `http://www.sics.se/dce/dive/dive.html`.

[httpEVER] EVERQUEST web site: `http://www.everquest.com`.

[httpHALF] HALF-LIFE web site: `http://www.sierrastudios.com/games/half-life/`.

[httpID] DOOM and QUAKE web site: `http://www.idsoftware.com`.

[httpNPS] NPSNET web site: `http://www.npsnet.nps.navy.mil/npsnet`.

[httpPARA] PARADISE web site: `http://www.dsg.stanford.edu/paradise.html`.

[httpUO] ULTIMA ONLINE web site: `http//www.uo.com`.

[IEEE93] Computer Society IEEE. Protocols for distributed simulation applications: Entity information and interaction. IEEE Standard 1278, 1993. Available at `http://www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame`.

[Kazm93] R. Kazman. Making WAVES: On the design of architectures for low-end distributed virtual environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 443–449, 1993.

[Kess96] G. D. Kessler and L. F. Hodges. A Network Communication Protocol for Distributed Virtual Environment Systems. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS'96)*, pages 214–221, 1996.

[Levo95] M. Levoy. Polygon-Assisted JPEG and MPEG Compression of Synthetic Images. *Computer Graphics (SIGGRAPH '95 Proceedings)*, Vol. 29, pages 21–28, 1995.

[Mace94] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environment. *Presence*, Vol. 3(4), pages 265–287, 1994.

[Mace95] M. R. Macedonia, M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham. Exploiting Reality with Multicast Groups. *IEEE Computer Graphics and Applications*, Vol. 15(5), pages 38–45, 1995.

[Mine95] M. Mine and H. Weber. Large Models for Virtual Environments: A Review of Work by the Architectural Walkthrough Project at UNC. *Presence*, Vol. 5(1), pages 136–145, 1995.

[Mogu84] J. C. Mogul. Broadcasting Internet Datagrams, 1984. Internet RFC 919, Information Sciences Institute, Marina Del Rey, CA. Available at `http://info.internet.isi.edu/in-notes/rfc/files/rfc919.txt`.

[Pand96] I. S. Pandzic, T. K. Capin, N. Magnenat-Thalmann, and D. Thalmann. Motor functions in the VLNET Body-Centered Network Virtual Environment. In *Proceedings of the EG Virtual Environments and Scientific Visualization Workshop'96*, pages 94–103, 1996.

[Pope89] A. Pope. The SIMNET network and protocols. Technical Report 7102. Cambridge, MA. Technical Report TR89-7102, BBN Systems and Technologies, 1989.

[Post80] J. Postel. User datagram protocol, 1980. Internet RFC 768, Information Sciences Institute, Marina Del Rey, CA. Available at `http://info.internet.isi.edu/in-notes/rfc/files/rfc768.txt`.

[Post81] J. Postel. Transmission control protocol - DARPA Internet program protocol specification, 1981. Internet RFC 793, Information Sciences Institute, Marina Del Rey, CA. Available at `http://info.internet.isi.edu/in-notes/rfc/files/rfc793.txt`.

[Schm97] D. Schmalstieg and R. F. Tobler. Exploiting coherence in 2.5-D visibility computation. *Computers & Graphics*, Vol. 21(1), pages 121–123, 1997. ISSN 0097-8493.

[Shaw93] C. Shaw and M. Green. The MR toolkit peers package and experiment. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 463–469, 1993.

[Silb94] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts, 4th edition*. Addison-Wesley, 1994.

[Sing94] G. Singh, L. Serra, W. Png, and H. Ng. BrickNet: A Software Toolkit for Network-Based Virtual Worlds. *Presence*, Vol. 3(1), pages 19–34, 1994.

[Sing95a] G. Singh, L. Serra, W. Png, A. Wong, and H. Ng. BrickNet - Sharing Object Behaviors on The Net. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'95)*, pages 19–25, 1995.

[Sing95b] S. Singhal and D. Cheriton. Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. *Presence*, Vol. 4(2), pages 169–194, 1995.

[Sing99] S. Singhal and M. Zyda. *Networked virtual environments: design and implementation*. Addison-Wesley and ACM Press, 1999. ISBN 0-201-32557-8.

[Snow94] D. N. Snowdon and A. J. West. AVIARY: Design Issues for Future Large-Scale Virtual Environments. *Presence*, Vol. 3(4), pages 288–308, 1994.

[Stal95] W. Stallings. *Operating systems*. Prentice-Hall, 1995. ISBN 0-02-415493-8.

[Stal96] W. Stallings. *Data and Computer Communications, 5th edition*. Prentice-Hall, 1996.

[Stev94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63346-9.

[Suda96] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. *Computer Graphics Forum (Proceedings of Eurographics '96)*, Vol. 15(3), pages 249–258, 1996. ISSN 1067-7055.

[Tane92] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. ISBN 0-13-588187-0.

[Wlok95] M. Wloka. Lag in Multiprocessor Virtual Reality. *Presence*, Vol. 4(1), pages 50–63, 1995.

Dipl.-Ing. Christian Faisstnauer
Horazstrasse 4/H
39100 Bozen
Italien

Mai 2001

Lebenslauf

12.August 1971:
geboren in Bozen, Italien, als Kind von Josef und Martha Faisstnauer.

September 1977 - Juni 1982:
Besuch der Grundschule "Johann Wolfgang v. Goethe" in Bozen.

September 1982 - Juni 1985:
Besuch der Mittelschule im "Franziskanergymnasium" in Bozen.

September 1985 - Juni 1990:
Besuch des naturwissenschaftlichen Realgymnasiums "R. von Klebelsberg" in Bozen.

Juli 1990:
Erfolgreiche Abschlußprüfung (Erlangen des Matura-Diploms).

Oktober 1991 - Oktober 1997:
Studium der Informatik an der Technischen Universität Wien.

November 1997:
Abschluß des Informatikstudiums als Diplomingenieur der Informatik mit Auszeichnung. Diplomarbeit "Navigation and Interaction in Virtual Environments" (in englischer Sprache).

Seit Dezember 1997:
Wissenschaftlicher Mitarbeiter am Institut für Computergraphik an der Technischen Universität Wien.

22.3.1999 - 2.4.1999:

10.4.2000 - 21.4.2000:

22.5.2001 - 5.6.2001:
Halten eines zweiwöchigen Programmierkurses (C++/ Java) in Brixen, Italien, im Rahmen eines Ausbildungsprogrammes für Netzwerkadministratoren, organisiert von der Landesregierung der Provinz Bozen, Italien.

Chris Faisstnauer
Via Orazio 4/H
39100 Bolzano
Italy

May 2001

Curriculum Vitae

12. August 1971:
born in Bolzano, Italy.

September 1977 - June 1982:
Primary school ”Johann Wolfgang v. Goethe” in Bolzano, Italy.

September 1982 - June 1985:
Secondary school at the ”Franziskanergymnasium” in Bolzano, Italy.

September 1985 - June 1990:
High school ”Raimund von Klebelsberg” in Bolzano, Italy.

July 1990:
Graduation from high school.

October 1991 - October 1997:
Studies in computer science at the Vienna University of Technology (TUW).

November 1997:
MS-Graduation from TUW with highest distinctions.
Diploma thesis: ”Navigation and Interaction in Virtual Environments”.

1997 - today:
Occupation as research assistant at TUW.

22. 3. 1999 - 2. 4. 1999:

10. 4. 2000 - 21. 4. 2000:

22. 5. 2001 - 5. 6. 2001:
Lecturer on C++/ Java (basic and advanced programming skills) in a private course organized by the local government of Bolzano, Italy, for the training and education of network administrators.

The Road goes ever on and on
Out from the door where it began.
Now far ahead the Road has gone,
Let others follow it who can!
Let them a journey new begin,
But I at last with weary feet
Will turn towards the lighted inn,
My evening-rest and sleep to meet.

J.R.R. Tolkien (1892-1973)
English poet and novelist.