uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Raed Ghannam Jarrar

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Image-based Rendering Protocols for Remote Interactive Walkthroughs on Mobile Devices

TITRE DE LA THÈSE / TITLE OF THESIS

A. Boukerche

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

E. Petriu

G. Wainer

W. Gueaieb

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Image-based Rendering Protocols for Remote Interactive Walkthroughs on Mobile Devices

by

Raed Ghannam Jarrar

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.Sc. degree in
Computer Science

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-59861-0
Our file Notre référence
ISBN: 978-0-494-59861-0

Abstract

The exploration of virtual environments in wireless mobile media devices has attracted the attention of researchers and developers mainly due to its potential applications in a variety of areas including entertainment, training, security, e-learning, etc. However, current technology of mobile devices lack the proper resources to handle complex and realistic 3D virtual environments. There has been a number of proposed ideas to solve this issue. The existing approaches use techniques that either employ limited user navigation modes or do not perform satisfactorily for interactive applications. In this thesis, we propose a new protocol that offers the user a richer navigation by pre-streaming the necessary imagery data to generate new views as the user wanders within the 3D environment. We introduce the idea of key partial panoramas, i.e., panorama segments that cover movements in any direction by simply strafing from an appropriate key partial panorama and streaming the amount of lost pixels. We have implemented our ideas and evaluated it against two well-known approaches. Experimental results show that our solution outperforms the selected approaches by minimizing the delay between image updates and by allowing a more complex navigation scheme than previous works.

Acknowledgements

I would like to acknowledge and extend my gratitude to the people whom with their help and support, I have made it here, writing these last couple of paragraphs in my master's thesis. Here I go with no particular order or preference...

Firstly my supervisor, *Professor. Azzedine Boukerche.* I appreciate all what you have done for me during the course of my graduate studies at the university of Ottawa; from the point you picked me, believed in me and in my potential, until this day. You have been a great mentor who always was there to support me and broaden my horizons. I now know what it means to do a job at your best and excel in it. I still remember when I promised not to let you down and to achieve something we are proud of. I hope I have fulfilled that promise. Your understanding, care and support will not be forgotten. Thank you for everything sir.

Next in turn are my lab comrades from the PARADISE lab. My research mentor and companion *Richard Pazzi* -I dropped the Dr. salute since we are friends!-. It has been more than a wonderful year and half since we started working together. We have shared all the hard work that resulted in this thesis; finding new ideas, discussing their feasibility, actually implementing the whole thing and all the hardships that have come along. I have to say that it has been wonderful working with you and that I truly believe that you are an extraordinary researcher and computer scientist. I sincerely thank you for your help, patience, enthusiasm and expertise you have granted me and I really hope we can continue in this work in the future. I think that we make a great research team my friend.

I would like to thank PARADISE lab veterans and friends; *Abdulaziz Al-Hamidi, Ahmad Shadid, Muhannad Al-Hajiry, Luqman Ahmad, Osama Abu Mansoor, Kaouther, Haifa, Yasmin, Annahit, Robson, Elie, Jeremy, Christiano* and all the remaining for their care and the beautiful days I have lived in the lab. I will never forget your encouragement and compassion. Abdulaziz, I still remember that day when we hunted those memory leaks down, on behalf of my system, we say thank you!

Muhannad, Osama, Richard, Haifa, Yasmin, Annahit, Robson and Christiano, thank you very much guys for attending my defense and making it one of the most charming days in my life. I will never forget this day and your great company.

On the personal side, I begin with my parents who have always loved me and believed in me. You are the reason I stand where I am today. I want you to be proud of your son and how you raised him and nurtured him with decency, morals and success. Every time I was down and thought this was not possible, I would only listen to your voices and despair would transform into determination. Thank you Mom and Dad, my greatest reward will be seeing you reading these lines. You will always be my number one.

Last but not least, all my friends and relatives who stood by me, whether I have known you from my home Jordan or here in Canada, I value your help and eagerness for my success. I thank you all and anyone who ever cared for me someday. Thank you all people. To you I will always be indebted.

Raed, December, 7th, 2008 : 10:12 PM, Ottawa. *(revised: March 11th, 2009 : 2:30 AM).*

To

My Mother and Father, Rasha, Reema, Mohammed and Ahmad

With sincere love and gratitude

List of Publications

The following publications by the author are relevant to the work in this thesis.

Journals:

- A. Boukerche, R. Jarrar, R. W. Pazzi. "A Predictive Image-based Technique to Support Virtual Environment Streaming.". To be submitted.

Conference papers:

- A. Boukerche, R. Jarrar, R. W. Pazzi. "An efficient protocol for remote virtual environment exploration on wireless mobile devices". *In Proceedings of the 4th ACM Workshop on Wireless Multimedia Networking and Performance Modeling* (Vancouver, British Columbia, Canada, October 27, 2008). pp 45-52.
- A. Boukerche, R. Jarrar, R. W. Pazzi. "A Novel Interactive Streaming Protocol for Virtual Environment Navigation through IBR". Accepted by the *IEEE International Conference on Communications (ICC)* 2009.

Contents

1 Introduction **1**

1.1 Motivations 4

1.2 Objectives 5

1.3 Contributions 5

1.4 Organization 6

2 Background and Related Work **9**

2.1 Multimedia Streaming over Wireless Networks 10

2.2 Networked 3D Virtual Environments Streaming Approaches 12

2.2.1 Geometry Replication 13

2.2.2 Progressive Meshes 14

2.2.3 Impostors 15

2.2.4 Image-based Model 15

2.3 Remote Rendering Frameworks 20

2.4 Virtual Environment Streaming Techniques to Reduce Network Load . . 20

2.5 Image-based Interactive Virtual Environments 23

2.6 Comparative Analysis 29

2.7 Summary 31

3 Architecture and Design of the Virtual Environment Streaming System 33

3.1 Virtual Environment Streaming System Architecture 34

3.2 Server Components Design 35
3.3 Mobile Client Components Design 36
3.4 Network Communication and Messaging 38
3.4.1 Standard Messages Format 39
3.4.2 Network Protocol Abstraction 41
3.4.3 Network Communication Modules 42
3.5 Server and Client Protocol Design 47
3.5.1 Server Protocol Design 47
3.5.2 Client Protocol Design 48
3.6 Summary 48

4 Image Response and Panorama on Demand Streaming Protocols 50
4.1 Image Response Protocol (IR) 51
4.1.1 IR protocol Implementation 51
4.1.2 IR protocol Analysis 54
4.2 Panorama on Demand Protocol (POD) 54
4.2.1 POD Protocol Implementation 55
4.2.2 POD Protocol Analysis 60
4.3 Summary 60

5 The Proposed Virtual Environment Streaming Protocol 62
5.1 PPP Protocol Objectives 63
5.2 User Navigation Scheme 64
5.3 3D Mathematics Background 64
5.3.1 View Frustum 65
5.3.2 Projection Plane 65
5.4 Partial Panoramas 67
5.5 Strafe Movement Streaming 69
5.6 Directional User Movement Streaming - Key Partial Panoramas 73

5.7 PPP Protocol Formalization 76
5.8 PPP Protocol Implementation 80
5.8.1 Partial Panoramas Algorithm 82
5.8.2 Depth Compression / Decompression 90
5.8.3 Panorama Data Streaming Mechanism 95
5.8.4 Current Implementation Status, Problems and Limitations 97
5.9 Summary 98

6 Performance Evaluation **100**
6.1 Experiments Hardware and Software Specification 101
6.2 Experiments Scenarios and Metrics 102
6.3 Experiments Framework 106
6.4 Performance Evaluation Results 108
6.4.1 Random Navigation Path Experiments 109
6.4.2 Rotation Weighted Path Experiments 111
6.4.3 Multiple Clients Experiments 113
6.4.4 Virtual Environment Complexity Experiments 115
6.4.5 Rotation Coverage Experiments 116
6.5 Summary 118

7 Conclusions **120**
7.1 Future Work 124

A Demo Application **126**

List of Tables

2.1 **IBR vs GBR on mobile devices** 29

2.2 **Image-based remote rendering approaches comparison** 32

5.1 Error percentage according to the step size. 75

6.1 **Experiment parameters.** 108

List of Figures

2.1 Progressive mesh representation. 14
2.2 Left: original 3D object. Right: texture mapped impostor object. 15
2.3 Unwrapped cylindrical panorama 19
2.4 Unwrapped cubic panorama. 19

3.1 Virtual Environment Streaming System Architecture. 34
3.2 Server Application Main Components. 35
3.3 Client Application Main Components. 37
3.4 *ServerMessage* and *ServerMessageCode* Class Diagram. 39
3.5 *ClientMessage* and *ClientMessageCode* Class Diagram. 40
3.6 Server network protocol abstraction classes. 41
3.7 Client network protocol abstraction classes. 42
3.8 Server network communication module. 43
3.9 Client network communication module. 44
3.10 Server protocol design. 47
3.11 Client protocol design 49

4.1 IR client sign in activity diagram. 52
4.2 IR movement/rotation activity diagram. 53
4.3 POD rotation activity diagram. 56
4.4 POD movement activity diagram. 57

5.1 View Frustum. 65
5.2 Projection plane and the focal length e. 66
5.3 (θ, β) rotation coverage. 68
5.4 Difference between old and new rectangles is streamed upon user rotation. 69
5.5 Strafe by D units from position A to position B. 70
5.6 Original and strafe generated images. 72
5.7 θ° rotation followed by a forward movement by D units to position B. . . 73
5.8 δ calculation. 74
5.9 PPP protocol explanation. 77
5.10 Unfolded cubic panorama. 81
5.11 Cubic panorama axis system. 81
5.12 Cubic panorama implementation. 82
5.13 Partial Panorama for current user view. 83
5.14 Extended cubic panorama representation. 83
5.15 Partial panorama on the extended panorama representation. 84
5.16 Partial panorama implementation classes. 85
5.17 Depth index compression into pixel data. 92
5.18 Depth compression/decompression implementation class. 95

6.1 Virtual environment streaming system screen shots. 102
6.2 Client application screen shots. 103
6.3 Movement script generator software. 104
6.4 Performance test client. 106
6.5 Performance test server. 107
6.6 Average waiting time and local cache hit ratio for a random path. 109
6.7 Average data traffic for a random path. 109
6.8 Average waiting time and local cache hit ratio for a rotation weighted path. 111
6.9 Average data traffic for a rotation weighted path. 111
6.10 Average waiting time and bandwidth utilization vs number of clients. . . 113

6.11 Execution time and rendering time percentage vs number of clients. . . . 114
6.12 Average waiting time and data traffic for different VE complexities. . . . 115
6.13 Average bandwidth utilization for different VE complexities. 116
6.14 Average waiting time and local cache hits for different *PPP* coverages. . 117
6.15 Average data traffic and bandwidth utilization for different *PPP* coverages. 118

A.1 Command center and first responder client software. 127
A.2 VE streaming system. Left: Second floor view. Right: Paradise lab. . . . 128
A.3 Left: Fire. Right: First responder. 128
A.4 Rescue process by first responder. 129
A.5 Rescue process by first responder. 129
A.6 Floor map. 130

Glossary of Terms

AOI Area Of Interest.

BPP Bits Per Pixel.

CPU Central Processing Unit.

FEC Forward Error Correction.

FOV Field Of View.

FPS Frames Per Second.

GBR Geometry-Based Rendering.

GPU Graphics Processing Unit.

HMD Head Mounted Display.

IBR Image-Based Rendering.

IR Image Response Protocol.

LAN Local Area Network.

LDA Loss Differentiation Algorithm.

LDI Layered Depth Image.

LOD Level Of Detail.

PDA Personal Digital Assistant.

PLC Packet Loss Classification.

POD Panorama On Demand Protocol.

PPP Partial Panorama Prediction Protocol.

PV Panoramic Video.

ROI Region of Interest.

RTCP Real-time Transport Control Protocol

RTP Real-time Transport Protocol.

SPLD Statistical Packet Loss Discrimination.

TCP Transmission Control Protocol.

TFRC TCP Friendly Rate Control Protocol.

UDP User Data-gram Protocol.

UML Unified Modeling Language.

VE Virtual Environment.

Chapter 1

Introduction

Multimedia communication for wireless mobile devices is in the spotlight nowadays mainly due to the advances we have witnessed in wireless technologies. Network speeds in general have been rapidly growing and wireless networks is not an exception. Today's mobile computing devices posses more powerful hardware too, enabling them to do things that were not possible a few years ago. The role of mobile devices is expanding largely in our world and it is expected that anything can be accessed and done using mobile devices. Streaming and sharing different kinds of content such as pictures, documents, and videos have become more popular than ever. The idea of immersing multiple avatars in a 3D world where each avatar simply uses a mobile phone or a light weight PDA has become possible with the evolution of todays technologies in terms of mobile computing and computer networks. Although we have witnessed significant hardware advances, mobile devices still lack the proper resources to render complex and realistic 3D virtual environments at interactive frame rates. Image-Based Rendering (IBR) is an approach that alleviates the complexity problem of virtual environments in constrained mobile devices. In IBR, images are used as the source for a scene renderer, thereby the processing task does not depend on the scene complexity, but on the image size only. IBR techniques can be combined with remote 3D rendering, i.e., the 3D rendering task is delegated to a server that renders and simply sends images to the client that displays

the image, in order to provide mobile devices with complex virtual environment applications. At its very primitive form, this hybrid approach usually wastes bandwidth, since streaming images for every frame is costly and results in network jitter and compromises the interactive experience. Several enhancements were proposed in previous works that tried to solve this problem. However, proposed navigation schemes in these works are very simplistic, limiting navigation and user interactivity.

Several applications are envisioned, for instance, remote exploration of touristic locations, virtual training, entertainment, and emergency preparedness and response, in which the first responder could wear a head mounted display that displays a virtual representation of a target area and is connected to a control center server which updates the responder's view as he/she changes position or as events occur. Augmented information about the target can be collected and displayed to the responders, giving them more information that helps them accomplish their missions.

There are many challenges to displaying 3D virtual environments to simultaneously connected mobile devices users. The system's response has to be quick to provide users with requested scenes, this is a realtime requirement that if failed to deliver, interactivity would be lost. Another challenge is the network load when large number of users are connected and accessing the system. Obviously, the network is the most constrained resource in this application. When the network is heavily loaded, response times are expected to increase. In addition, mobile device users should be granted an advanced navigation scheme to better explore the environment. Limiting navigation options and using simplistic movement due to constrained resources negatively impacts user's experience with the 3D environment and demolishes the point of using mobile devices to access remote 3D virtual environments. Any proposed solution has to consider these challenges in its design and try to find ways to overcome them or at least minimize system's dependency on them.

In this thesis, we discuss some techniques and propose protocols that allow for more complex user navigation mimicking real human movement, and therefore can be applied

to highly interactive applications of this idea such as emergency preparedness mentioned above. Another extremely important factor is that the new proposed protocol delivers requested scenes to the client more quickly than existing solutions; it lifts some of the load of the network and moves it to the mobile device. The mobile device is given information that can be used to generate next requested views or at least a major part of them. This minimizes delay times and dependency on the network and therefore provides a more interactive and realistic virtual environment to the mobile device user.

The proposed scheme is based on a client-server architecture. The server maintains the 3D virtual environment and its states as well as user's status. Mobile devices clients communicate with the server informing it of events they encounter. The server handles the rendering of new scenes and sends them to the client. We use panoramic views at the client. A panorama is an image that captures all the views from a certain position in the 3D environment, enabling the user to look freely from that position. Our scheme does not send full panoramas that cover all views since most of the time, users will not need to use the entire view at a position. Instead it sends parts of the views predicted to be used by the client.

The proposed protocol takes advantage of IBR techniques that allow the client to generate new views from the existing view the client currently has. Using very simple but yet effective trigonometry, a special movement called *strafe* where the user moves horizontally with out rotating can be generated from the client's current view. Most of the new scene can be locally generated at the client and thereby minimizing the delay between image updates at the client end.

This efficient solution of the strafe movement is generalized by the protocol to include any movement type. Yes any movement in any direction. By using a set of key images positioned carefully around the user's position, these key images are used as strafe movement sources. Any movement can be generated by strafing from the appropriate key image. This simplifies the task of predicting user's future movements. For complex navigation patterns, existing solutions that rely on basic movements and send all possible

next views is simply not feasible to apply since the number of possibilities is huge in the new navigation scheme. By grouping movements in different directions under a certain key image we can predict user movement in those directions by sending a key image that generates scenes for these positions if the user navigates there.

Using these techniques, the mobile device is not completely bound to network and server delays once it needs new scenes. It can locally generate next scenes or in worst case most of them and hence minimize network load and dependency and promote response times. Our techniques simply outperform existing solutions by caching images predicted to be used in future user movements either directly or as sources to generate these future views on the client mobile device.

In this thesis, we developed a virtual environment streaming system that incorporates a server and mobile clients. Our proposed protocol was implemented and compared against two other protocols that most of the existing solutions we found in the literature fit into. Results prove that the new idea outperforms the existing solution and promises for more interactive experiences and to enable more complex applications to be used on mobile devices in the future.

1.1 Motivations

The most important motivation of this thesis is to enable the exploration of complex and compelling 3D virtual environments on mobile devices. Providing an efficient streaming solution would open the door to many advanced applications that can be extended to be used on mobile devices. This is our vision in this thesis. Other motivations include that current solutions we found in the literature do not offer mobile device users with free movement patterns which limits interactivity and consequently limits applications in high need of such complex navigation. Slow response times that do not allow interactive frame rates also motivated us to develop a solution that performs faster and tries to achieve interactive frame rates. Lack of prediction of complex movements also inspired

us to research and find ways of predicting future needed views and make them available on the mobile client prior to needing them in order to boost the system's performance.

1.2 Objectives

The objectives of this thesis are listed below:

1. Develop a technique that enables 3D environment exploration on mobile devices at interactive rates.

2. Investigate and design a mechanism that provides mobile device users with complex movement options that mimic real life movement of a human and found in 3D games that run on stand-alone computers.

3. Develop a prediction scheme that predicts future user movements for this complex movement pattern and make them available on the mobile client prior to needing them.

4. The proposed system should relieve the mobile client from totally depending on the network and server processing delays when new scenes are required and thereby providing a more scalable streaming system and more independent mobile devices.

1.3 Contributions

During the course of work in this thesis, we have achieved the following contributions:

1. Developed and implemented a virtual environment streaming system that supports multiple streaming protocols and simultaneously connected mobile devices, and uses different network protocols such as TCP and UDP.

2. Developed a new virtual environment streaming protocol, called partial panoramic prediction protocol *(PPP)* which allows complex navigation of the environment

and outperform existing ideas. It mainly relies on predicting user movements and caching it on the client in advance. The new protocol is also adaptive to user movements unlike existing ones which only react to user movement. It learns from user movements and changes its parameters to suit the user's movement pattern and improve performance. *(PPP)* protocol includes new ideas to achieve that which are listed below:

(a) Partial panoramas: save the streaming of unnecessary data and is used to predict views at the current position that will be most likely used.

(b) strafe movement streaming: an efficient method to generate new scenes for a strafe movement from the existing view on the client.

(c) Key partial panoramas: carefully positioned set of partial panoramas used as sources of strafes to generate scenes to any movement the user might take.

(d) Depth compression: an indexing idea that compresses pixel depths and participates in improving the performance of the protocol.

3. Developed and implemented two streaming protocols that most existing solutions fall into. These protocols are the image response *(IR)* and panorama on demand *(POD)*. *IR* protocol sends images of new views once requested and *POD* protocol sends a complete panorama for each position the client visits.

4. A comprehensive and analytical study of existing virtual environment streaming techniques.

1.4 Organization

The remainder of this thesis is organized as follows:

- **Chapter 2: Background and Related Work.**
  This chapter introduces the reader to background information necessary to under-

stand the scope of this thesis and the current state of the research in this topic. Distributed virtual environment techniques including image-based rendering and other approaches are described. More emphasis is put onto the IBR model and its different rendering algorithms. Panoramas and their different types are described in depth. Remote rendering frameworks and bandwidth saving ideas are explained next. Finally, a review of existing image-based rendering solutions for wireless mobile devices is provided and each idea is analyzed and criticized.

- **Chapter 3: Architecture and Design of the Virtual Environment Remote Streaming System.**

  This chapter describes the architecture and design of the 3D environment streaming system we developed. It identifies the architecture of the system and how the server and mobile client interact together. The software components of the server and the mobile client and how they collaborate together are then thoroughly explained. Aspects such as network and messaging implementation details and protocol design are discussed as well.

- **Chapter 4: Image Response and Panorama on Demand Streaming Protocols.**

  This chapter explains two categories of virtual environment streaming solutions. Image response *(IR)* and panorama on demand *(POD)*. Most of the existing IBR solutions can be grouped under these two protocols. These protocol's details and our implementation are then explained. Furthermore, the protocols' behavior is analyzed; strengths and weaknesses of each is identified as we set our motivation for the new protocol, *PPP* .

- **Chapter 5: Partial Panorama Prediction Virtual Environment Streaming Protocol.**

  This chapter introduces the new proposed streaming protocol, *PPP* . It sets the objectives of the protocol and defines the movement scheme users are offered under

this streaming protocol. The ideas of partial panoramas, minimum rotation coverage, strafing, key partial panoramas are then explained in great detail. Then the protocol is formalized, server and client algorithms are formally written. Lastly, our implementation of the protocol is explained, details of the new ideas such as partial panorama implementation, depth compression and types of panoramas used are described. The chapter concludes with the current implementation status and limitations.

- **Chapter 6: Performance Evaluation.**

  This chapter presents the experiments we conducted on the protocols we developed. First, hardware and software configuration is specified. Then, experiments scenarios and their factors are explained as well as the metrics used to measure the protocols. We developed a special testing framework to execute the tests, this framework is explained too. Results of the experiments are then shown for many different scenarios. We comment and analyze the results and identify limitations and problems in the protocols and their implementations.

- **Chapter 7: Conclusions.**

  This chapter summarizes what has been done in this thesis and identifies what has been accomplished, what problems still exist, and sets future research directions.

- **Appendix A: Demo Application.**

  An emergency preparedness application built on top of the virtual environment streaming system developed in this thesis is presented as a sample application of the ideas we developed.

Chapter 2

Background and Related Work

In this chapter, we introduce the reader to background information concerning remote 3D virtual environment navigation and explain existing approaches and solutions that try to tackle this topic. In section 2.1, we talk about media streaming in general over wireless networks and its different issues. We differentiate between interactive and non interactive media and show that streaming interactive media, where the user participates in its progress and flow, is more difficult and requires different solutions to be employed. Section 2.2 describes networked 3D virtual environments and existing approaches that stream these environments to a remote client. We compare between these approaches and judge their suitability for use on light weight mobile devices. We then expand on the image-based model and show its existing techniques; more emphasizing on a particular IBR technique, *panoramas*. We shall define what a panorama is and what it is used for as well as its different formats and compare between them. Sections 2.3 and 2.4 show existing remote rendering frameworks and techniques used to minimize VE streaming. In section 2.5, we discuss existing IBR approaches that try to solve the problem of displaying a 3D virtual environment on mobile devices which we found in the literature. In section 2.6, we compare between these approaches and identify their advantages and shortcomings. We set our motivation in this thesis based on the current difficulties and limitations found in existing approaches and strive to solve them.

2.1 Multimedia Streaming over Wireless Networks

There are two factors involved in streaming multimedia; the network medium on which the media is transmitted and the type of the media itself. Different types of media and networks require different ways to deal with their streaming.

The nature of the transmission medium imposes new issues and challenges to be dealt with. Wireless networks have different characteristics from wired ones. Their bandwidth is more limited and time varying and they suffer from high error rates and interference which cause packet losses. These issues have lead to solutions specifically tailored for wireless networks. To solve the problem of packet loss in wireless networks; error control mechanisms were developed. For example, Forward error correction (FEC) duplicates important data in order to be used on the client to recover from errors. Another form is to make the receiver notify the sender of lost packets to resend them.

Multimedia such as video, require high bandwidth. Therefore, bandwidth utilization has to be maximized to provide a better quality of service. Rate control mechanisms were introduced to achieve this requirement. The *Transport Control Protocol (TCP)* [63] is designed for reliable data transmission over wired networks. TCP is not suitable to be used in wireless networks however because of its congestion control mechanism. TCP recovers from congestion by decreasing transmission rate to 50% which affects the bandwidth utilization and therefore makes TCP unsuitable for continuous media streaming. The *User Datagram Protocol (UDP)* was designed to support continuous multimedia streaming but does not implement any rate control mechanism. *Real-time Transport Protocol (RTP)* [64] and its control protocol (RTCP) aim to support continuous media transmission but does not guarantee quality of service. *TCP Friendly Rate Control Protocol (TFRC)* [65, 66] is widely used for wired networks to handle congestions.

However, rate control on wireless networks is a different matter. The task of a rate control protocol is to reduce the transmission rate in case of packet losses in the network due to congestion. In wireless networks, packets might be lost due to congestion or due

to wireless problems such as interference. In the later case, the solutions designed for wired networks are unable to determine the reason of the packet loss and assumes it is due to congestion. Therefore, sending rate is reduced when it should not, and streaming performance decreases. Therefore, a way of differentiating the cause of the loss has to be incorporated into the rate control mechanism. Several Loss Differentiation Algorithms (LDAs) exist in the literature, such as: *Biaz* [67], *mBiaz* [68], *Statistical Packet Loss Discrimination (SPLD)* [69], and *Packet Loss Classification (PLC)* [70].

Concerning the media factor, we can divide multimedia into two broad categories: *interactive* and *non-interactive.* Non interactive media is linear and continuous; meaning that it does not involve any user control. The media to be streamed is always fixed and known in advance. Obvious examples are video, music, images and simple text. On the other hand, interactive multimedia enables the user to interact with the flow of the media and therefore is non-deterministic. Navigating virtual environments is a good example of this category. The user can navigate within the environment, therefore affecting the media to be streamed.

Streaming non-interactive media such as videos is *predictable.* The sequence of the frames is known in advance and is always constant. Therefore streaming protocols designed for such media can use buffers on the client to accumulate frames to avoid jitter during playback since the server already knows the data to be streamed. These scheduling and buffering techniques can not be used with interactive media such as interactive virtual environments however, since the server can not perfectly predict where the user will navigate in the environment. This task is not as trivial as transmitting constant video frames. Some solutions limit the user's navigation capabilities in the environment to a very basic movement scheme so that it can gather all the possible points the user might navigate to from his current location, and stream them in advance to the recipient.

The nature of the client that will receive the virtual environment scenes is very important as well. Mobile clients such as PDAs, smart phones or head mounted displays (HMDs) possess low processing capabilities, small storage and primitive 3D graphics

accelerators. Consequently, only simple 3D scenes can be run on these clients. The format of the transmitted scenes therefore is crucial to enabling these clients to display visually rich scenes.

In this thesis, we do **not** concentrate on solving wireless networks issues discussed above such as rate control or error control. We strive to find ideas that enable the prediction of user's navigation in a virtual environment given a complex navigation scheme, and stream predicted scenes *efficiently* and *in advance* to a mobile client. Therefore reducing delay times, minimizing network jitter and providing more complex navigation. Another goal is to stream these scenes in a format that isolates the complexity of the virtual environment from the mobile client. Thereby, enabling mobile devices to navigate visually complex 3D virtual environments that normally can not be run on the mobile device due to its small processing and graphical power.

In the next section, we discuss different ways used to stream virtual environment scenes and determine which among them is more suitable for use on the light weight mobile device.

2.2 Networked 3D Virtual Environments Streaming Approaches

There are many applications that require a 3D virtual environment to be shared over a network. These applications include but are not limited to: remote walk-throughs; where a remote user navigates the virtual environment, networked games played online or on a local area network (LAN); where each player interacts with the environment and other players, and collaborative virtual environments that simulate a real world scenario and used for training for example. Enabling remote clients to access and interact with a virtual environment residing on a server requires that the servers and clients use a common format to represent scenes in the virtual environment[2, 3, 4]. We can assume that the most constrained resource in a networked 3D virtual environment is the network it-

self. Network transmission rates are considerably slower than CPU processing speeds and graphics processing units (GPUs) optimized rendering capabilities. Therefore, virtual environment streaming solutions have to focus on minimizing bandwidth consumption as much as possible without compromising the quality of the scenes delivered to the client. Another factor to be considered in the case of mobile clients is their low processing and graphics rendering power. Streaming solutions have to stream the virtual environment in a format which is easy to render on the client given its constrained resources. Downloading 3D models to a client depends on the size of the model and the bandwidth of the network. Today's realistic 3D models are made of hundreds of thousands of vertices and a not inconsiderable number of textures images. Therefore rendering a complex virtual environment on a mobile device is not a straight forward process. The literature includes a number of distributed graphics models, among them we discuss: geometry replication, progressive meshes, impostors and the image-based model.

2.2.1 Geometry Replication

In this approach, a copy of the 3D models' geometry is stored on the client in order to be rendered by the local client hardware. The geometry can either be stored on a persistent storage media on the client such as hard disks or optical drives as done by computer games like DOOM 3 [5] and GTR [6], or downloaded from the server once the client needs it, such as VRML [27] browsers. Multi-player games store the geometry locally on each client. The client performs all 3D rendering. Only events related to the game state are spread across the network to ensure game consistency among all client machines. For example, when a player shoots another player; all players: the shooter, the deceased and the witnesses have to be updated and informed about what happened. If we try to apply this technique to our case of interest; a low bandwidth wireless network and light weight mobile clients, we will find that this technique is simply infeasible. Firstly, the size of realistic 3D models is large and takes long times to download. This is unacceptable in cases where the client performs an action, i.e. move or shoot, and expects to see the result

in real time. This process has to be repeated frequently, at arbitrary times, and under a realtime constraint. Assuming that we tolerate this problem; mobile clients are still unable to render the received 3D models in the same quality as rendered by a dedicated server. Therefore, this technique does not well suit exploration of large, continuous and complex virtual worlds.

2.2.2 Progressive Meshes

Progressive meshes store the simplest representation of a model alongside detail records used to incrementally render the simple mesh exactly as the original mesh model [7, 8, 9, 10]. A low polygon version of the original 3D model is first streamed, then instructions on how to create more complex representations are progressively streamed. Figure 2.1 depicts this process. The mesh is enhanced from 171 faces progressively to its original 10056 faces. This technique allows progressive streaming of the model which enables the client to view the model earlier -in lower quality- and progressively refine the model until it is restored to its original quality. On the downside, progressive mesh generation is complex and takes long times. In addition, the 3D model to be streamed has to fit entirely into the memory of the mobile client. This means that large and complex 3D models can not be streamed to mobile devices using this technique.

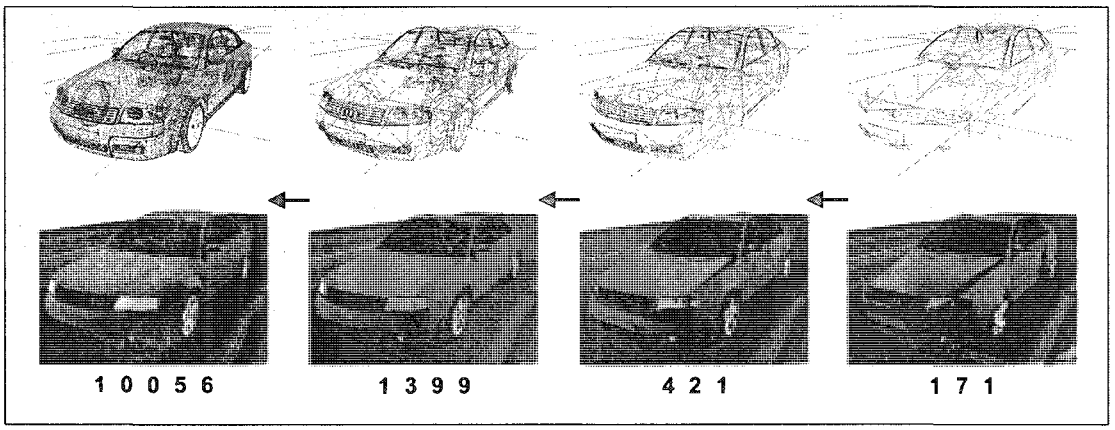


Figure 2.1: Progressive mesh representation.

2.2.3 Impostors

Impostors illude their viewer that he/she is seeing a complex 3D model although actually they see a simple shape wrapped with the appropriate image. In this technique, the server renders the complex 3D model into an image which is sent to the client. The client then texture map this image to a simple shape such as a plane or a box [11, 12, 13, 14]. Figure 2.2 shows how impostors work. A complex 3D model is replaced by a simple texture mapped polygon facing the viewer that cannot be distinguished from the original object from the proper viewpoint. Since rendering a simple object such as a plane or a box is much faster than rendering the actual complex 3D model, impostors can be used to render objects that are far from the user's sight. This results in faster rendering times and lower bandwidth usage, since the size of an impostor is considerably less than an actual 3D model. This technique is not as good for objects close to the user, since the user can notice the lack of depth in the impostor. Another point is that if the user changes his viewpoint, i.e. rotates or moves, a new impostor relative to the updated viewpoint has to be streamed to the client.

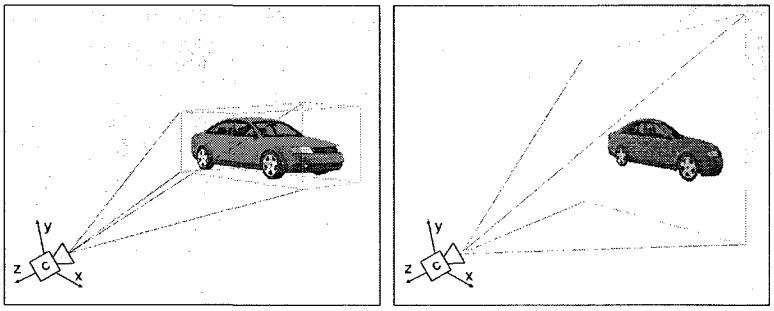


Figure 2.2: Left: original 3D object. Right: texture mapped impostor object.

2.2.4 Image-based Model

Realistic 3D models are rather large in size making them unsuitable for low bandwidth networks. Their rendering is heavy and require computers with powerful specifications, and therefore, light-weight mobile devices can not render them. The image-based model

maneuvers these difficulties by representing the virtual environment using images instead of geometry. In image-based rendering (IBR), the server - equipped with powerful rendering hardware - takes care of the rendering task. The server sends resultant images to the client which displays the image. This concept is sometimes called *Remote Rendering.* The client's task has transformed from rendering the complex geometry into simply displaying received images. This task is much simpler and its use is practically supported on today's mobile devices.

There exist many IBR techniques that use a set of reference images to construct intermediate views. These techniques can be used on the client to avoid long waiting times for new images. Images in the local client cache can be used to generate the new views or at least a portion of them until new reference images arrive from the server. IBR can render photo-realistic scenes from the virtual environment that does not depend on the complexity of the scene. Geometry-based rendering (GBR) [15] on the other hand, relies on projections and 3D mathematics to render the virtual environment, which makes it dependent on the complexity of the scene in terms of the number of vertices, texture map quality, lights, shading effects and so on. Therefore, only powerful computers can render complex virtual environment in realtime using GBR. IBR however, depends on the size of the reference images which eliminates the complexity of the virtual environment from being a factor in its use, and thus effectively allows low end computers such as mobile devices to display the most complex of virtual environments that were practically impossible to render using GBR. All IBR techniques are based on the plenoptic function introduced by Adelson and Bergen [16]. In their work, the world can be seen as a set of light rays filling the space that can be seen by eyes or cameras. The plenoptic function represents these rays. The plenoptic function $P(\phi, \theta, \lambda, t, V_x, V_y, V_z)$, is a 7D function with an eye or a camera at all possible positions in the world (V_x, V_y, V_z) that records the intensity of the light rays passing through the center of the camera at every possible angle (ϕ, θ), for every wavelength λ, at every time t.

In their survey on image-based rendering techniques [28], authors categorize IBR

techniques according to the intensity of geometry used into the following categories: rendering without geometry, rendering with implicit geometry and rendering with explicit geometry. All of these techniques implement the plenoptic function or a subset of it given some restrictions on the camera's movement. Examples of techniques that do not use geometry in rendering are lightfield and lumigraph [17, 18] which are 4D subfunctions of the Plenoptic function that assume that the scene is static. Such representations have a large amount of data and high computational complexity. The image acquisition systems for these representations are generally complex camera arrays, such as the Carnegie Mellon mobile camera array [19] and the Stanford multi-camera array [20].

Rendering with implicit geometry uses a number of images taken at correspondent positions to generate new views. No direct 3D information is included in this model; only information gathered from the reference pictures positions is used to render new views. In the view interpolation technique [29], new views can be generated from two reference images. The position of the reference images greatly impacts the quality of the resulting views. The view morphing algorithm [30] generates arbitrary viewpoints on the line between the optical centers of two reference images.

Techniques with explicit geometry use 3D information linked with images such as pixel depths and coordinates. In his thesis [31], McMillan introduced the 3D warping technique. Provided that the depth of each pixel in input images is known and given a viewing matrix for input images and the new images; pixels from the input image can be projected into 3D positions using the input image viewing matrix and pixel depths and re-projected back to the new image according to its viewing matrix. This method usually generates holes in the output image. techniques such as drawing each pixel as a circle rather than a pixel or splatting [32, 33] can be used to fix this problem. Another problem is occlusion caused by views visible from the new image that where not visible in the input image. Layered depth images (LDI) [34] was proposed to solve this problem by encoding a list of depth and color values into each pixel in the input image so that new visible points which where hidden in the input images can be fetched from this list.

More advanced techniques such as LDI trees where proposed to better solve the occlusion problem as well.

All the aforementioned techniques try to generate intermediate views from a set of input images. For the purposes of this thesis, we will be using panoramas to render views in any direction from a certain point and incorporate some geometrical information into the panorama representation we use. This information will be vital for generating new views the user of the virtual environment might navigate to. This will be explained in details later in this thesis when we propose the new virtual environment streaming protocol. Panoramas are explained next in detail.

Panoramas

The plenotpic function represents real world views. the user can be at any position, look anywhere, any time and in any lighting conditions. In virtual environments however, the user experience is more limited, depending on the application and the available hardware. Subfunctions of the 7D plenoptic function are used to constrain the viewing space. For example, if we assume the light is constant, then λ can be omitted and we get a 6D plenoptic subfunction. If user movement is limited to a single plane, then the plenoptic function can then be reduced by one dimension making the viewing space 2D. Assuming that the user does not move, panoramic images can be constructed by horizontally rotating a camera in a certain point in the world.

In [25], a panorama is defined as any wide view of a physical space. A panorama captures visible view from a point in space in all directions. A panoramic image is normally projected on a 3D shape to give the user the illusion of seeing new views. Types of shapes used today include spheres, cylinders and cubes. Figure 2.3 shows an unwrapped cylindrical panorama to a flat image. Panoramic images are now popular 3D IBR representations due to QuickTime VR [21] and others [22, 23]. Spherical panoramas cover 360 degrees horizontally vertically. This enables the user to look any where from where he is standing. The disadvantages of this type of panoramas are that it creates

distortions in the image and that it is inefficient to render since it requires multiple rendering passes. Cubic panorama is another IBR representation where a panorama captures 360° field of view horizontally and vertically, and stores the result in six faces of a cube. This is a type of projection for mapping a portion of the surface of a sphere (or the whole sphere) to flat images. Four cube faces cover front, right, back and left, one the top and one the base, each of them having $90° \times 90°$ field of view. Panoramic images in the cubic projection are commonly used as the image source by several spherical panorama viewers such as [21]. Cubic panoramas generation is more efficient and is supported by most of today's standard graphics cards [26]. Figure 2.4 shows unwrapped cubic panorama faces.

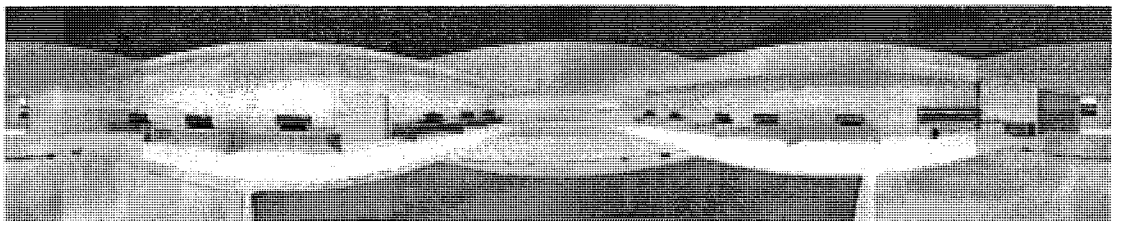


Figure 2.3: Unwrapped cylindrical panorama

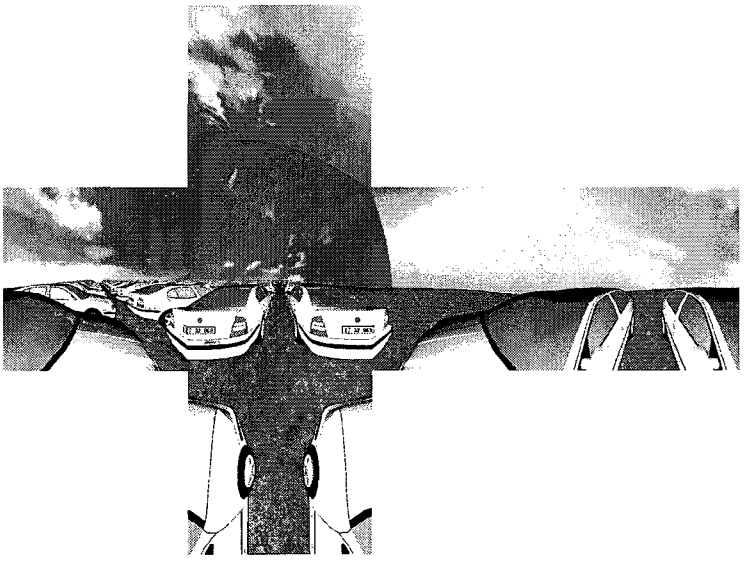


Figure 2.4: Unwrapped cubic panorama.

2.3 Remote Rendering Frameworks

Several remote rendering systems and frameworks have been developed. A very simple form of remote rendering is remote desktop applications, such as Virtual Network Computing (VNC) [36] or Microsoft Remote Desktop. In VNC, a system that runs a VNC server sends images captured from its screen to remote clients. The client triggers the updates by using the mouse or the keyboard. These input events are detected and sent to the VNC server which duplicates them locally and sends the resulting images back to the client. Chromium [37] is another remote rendering framework. It uses a modified implementation of OpenGL [72] that enables applications to use functions for parallel and cluster graphics rendering. Chromium intercepts the application's OpenGL calls sends them for rendering to a graph of processing nodes. The resulting images are sent to a remote client for display. In [38], the server renders the virtual environment as a movie which is then sent to mobile clients. This solution does not allow user interaction with the environment. In the system presented in [39], the client uses a web browser to display the remote 3D virtual environment. Authors state that they provide better frame rates using IBR methods. The system in [40] combines the processing power of the clients and the server. The client only renders 2D line primitives which have been generated on the server and derived from arbitrary 3D scenes. In [41], the authors developed a remote control interface for SGI's Open Inventor [42] applications. Compressed images are sent from the server to a Java-based client. The client sends events through CORBA [71] requests.

2.4 Virtual Environment Streaming Techniques to Reduce Network Load

Several techniques have been developed to handle network bottlenecks in distributed virtual environments. Virtual environment streaming requires a large amount of geometry

and high resolution textures to be transmitted in realtime. Area of interest (AOI)[43] also known as region of interest (ROI), is one of the most popular techniques used to reduce the network load of networked virtual environments. AOI tries to reduce the amount of data processed and streamed across the network by ruling out data that is not interesting to the user at an instance of time. Usually a user interacts within a small area of the virtual environment which is called the host area of interest. If only objects in that area are processed and streamed to the client, a significant amount of processing time and network load can be saved. Objects and areas that are far from the user's viewpoint are assigned lower priorities for rendering and streaming.

The filtering technique presented in an AOI-based approach can be given by the distance to the viewer or by visibility. In the first technique, the virtual environment is divided into regions. The user receives only objects that are positioned in immediately adjacent regions. Objects constantly send updates to the region in which they are located which allows the hosts in that area to be notified when an object moves [44]. In the other technique, information is filtered by visibility to the user, which saves processing and network bandwidth by only streaming updates of objects that are within the user's field of view. The visibility can either be predetermined [45], or specified by a camera or view frustum.

The AOI technique can be accompanied by another technique called *multi-resolution* or *view dependent simplification* [73]. The work in [74] for example, proposes a client-server system where only geometry that is visible to the client is extracted and sent to the client. The client receives the geometry and renders it using a specialized embedded system version of OpenGL, OpenGL ES [75]. View dependent simplification changes the resolution of 3D models depending on factors such as the user's location, manner of navigation, illumination and the scene's rate of change. The 3D models are constantly converted to the right level of detail according to how these parameters change. This technique however increases the size of the 3D models and require the whole model to be stored in the main memory. For 3D models larger than the mobile client's main memory,

solutions such as external memory, also called out-of-core [76] stores the geometry on the disk. This can lead to serious performance hits since disk access speed is very slow compared to CPU speed and therefore accessing the disk constantly -almost every frame- degrades the rendering performance on the mobile device.

Another technique that reduces the virtual environment information streamed over the network is dead reckoning. In dead reckoning, a host knows the behavioral model of the remote objects and it predicts the actual state of the object based on the model and previous states. The same dead reckoning prediction is run on every remote host on the local objects to detect the prediction accuracy. Prediction parameters are usually limited to object speed, position and direction [46]. Most of the techniques are based on the fact that the resolution of the displays and the eyes restricts the amount of perceived detail. Following this fact, objects that fall within the user's field of view may have different relevance to the user. Therefore, different Levels of Detail (LOD) can be used to exploit this concept by providing different object representations in terms of complexity (number of polygons, texture resolution, etc). This allows a progressive transmission of objects based on the user's perception and available bandwidth. As discussed earlier, progressive mesh [7] is a good example of a LOD-based technique.

An alternative method presented in [77] is called Remote Line Rendering. The rendering process is shared between the server and the client. The server streams 2D line primitives to the client which renders them. This method aims to reduce the network traffic and split the rendering work more fairly between the client and the server. The server maintains the virtual environment and is responsible for converting the 3D models into 2D lines to be sent to the client. The server extracts feature lines from the model and projects them to 2D window coordinates and sends them to the client. The client informs the server of user movements and renders the 2D primitives upon arrival. This technique suffers from limitations due to using 2D lines. Fully shaded and colored objects cannot be rendered on the client. The client can only render feature lines of the objects. This solution does not offer full 3D featured environments to the client.

2.5 Image-based Interactive Virtual Environments

There has been several attempts to provide virtual exploration of remote locations through image-based rendering.

The work in [78], uses the chromium architecture [37] to render virtual environment scenes on a cluster of workstations. Each workstation renders a part of the image and the final image is then assembled, encoded and streamed to the client over a TCP channel on a wireless network. This idea resembles the simplest form of IBR use for networked virtual environments. The requested images are generated on demand i.e. once the client actually changes his view. This technique neither considers user movement patterns nor predicts or buffers anticipated images in advance. The rendering process is boosted on the server using clustering techniques, but no buffering or user navigation awareness is offered.

Bradley et. al. [47] have proposed a panoramic image-based navigation system for remote exploration of real environments. Their proposed system utilizes a special camera to capture overlapping pictures of the vicinity that are mapped into a sphere. These images are transformed into a cubic panorama by projecting the spherical image into a cube. In their approach, the virtual environment is considered as a number of predefined locations connected by a graph, in which users can navigate to adjacent nodes. A user is able to look anywhere within a virtual panorama, while the system displays the available predefined navigation routes to the user. When a user moves to one of the predefined locations, the system retrieves the corresponding panorama. Since the panoramas are constructed from photos of the environment, this solution provides a photo-realistic navigation experience. However, the user movement is limited to predefined routes, which negatively affects interactivity and requires the acquisition of several panoramas in order to avoid noticeable breaks or jumps in the navigation.

The remote walkthrough system presented in [48] uses two rendering methods: panorama and morphing. Thousands of pictures were taken from a city using eight video cameras

and then projected into a cylindrical panorama. Each cylindrical panorama represents a location in the city. An algorithm based on Seitz and Dyer's view morphing algorithm [30] was used to solve the problem of displaying images at multiple arbitrary locations. The morphing algorithm needs a triangular region, whose vertices are original images, to generate intermediate images at any location within the triangle. This is done by interpolating positions and colors between original images. The specification of the vertices of the triangles is manual. Users have to specify the original images that form the triangles. The authors also mention that they used a variation of the view morphing algorithm to minimize distortions in the resulting images. In their approach, a user can move forward, backward and rotate the view. The system switches between the panorama and view morphing methods according to the user's movement and position. This system provides a better navigation scheme than the one proposed in [47], since it can generate views from theoretically any point. The problem is that reference images to be used by the morphing algorithms have to be specified manually, which is a cumbersome and arduous task. In addition, generated images still suffer from distortions due to view morphing calculations, and the quality of these images depends on the reference images specified by the user.

The Rail-Track Viewer [51] focuses on using Image-Based Rendering (IBR) techniques to explore large and complex virtual scenes on a remote high resolution display across the Internet. The authors developed a system that allows the user to move forward and backward by restricting the movement along pre-selected "tracks", in which the user can interact with the environment only at some points as a panoramic viewer. Basically, data sets are pre-rendered using appropriate rendering engines and the images and their geometric information are stored on a server. The reference images are sent to a client on request. The client will use reference images to reconstruct novel views by using a layered panoramic IBR model. The predefined locations closer to the viewpoint in the virtual environment will have more panoramic information cached. The client will maintain at least two points with complete panoramic imagery, which requires large bandwidth

and storage capacities. According to the authors, experiment results have shown that their system is able to render at 10 to 15 frames per second (FPS) for 1k x 1k image resolution. As a drawback, this work cannot be directly applied to thin mobile devices due to high bandwidth and storage requirements. In addition, user navigation is limited to pre-selected tracks and viewpoints.

A client-server approach to image-based rendering on mobile terminals is presented in [50], whose objective is to make it possible to render 3D scenes at interactive frame rates on mobile devices through an IBR method and a client/server architecture. The author's main contribution is how to place the cameras in the virtual environment in order to avoid exposure and occlusion problems when using IBR. They use the footprint of an urban environment (streets and buildings) to properly select reference images to be transmitted to the mobile device. Basically, the 3D virtual environment is rendered on the server, which uses the camera placement algorithm to capture pertinent reference images, and then sends the images to the client that renders new views using IBR. The server sends the initial reference images and the client starts its IBR process. Experiment results have shown that their system is able of rendering less than 4 FPS on a PDA and between 8 to 20 FPS on a PC.

Yu et. al. [49] use an image-based approach to provide remote walkthrough on mobile devices. Cubic panoramas are organized into panoramic videos (PV) according to their spatial positions. One PV can be represented as a path in the environment. The user can move along the designated paths. In their system, a server stores the captured panoramas and organizes them in PVs. The server maintains a list of parameters for each client, such as user's viewpoint, moving manner and cache status. Required image segments are determined by these parameters and then sent to the client. The client notifies the server of any change of its status. Data caches are used on both the client and the server. On the server side, the data nearest to the user's current position and the most probable wandering path is fetched from the disk. On the client end, image segments in the vicinity of the user's position are also cashed. A rate control scheme is employed in

which the server checks the available bandwidth periodically. Depending on the available bandwidth, the server adapts the size of image segments and increases the user step size to skip image segments if needed. The drawbacks of this solution include the limited interactive experience, since the movement scheme is tied to certain locations, and the prefetching scheme is not well presented, it is not clear how their solution computes the most probable paths the user will take within the environment.

Another system that uses PVs is PanoWalk [80]. Scene data is compressed by a customized mobile device MPEG encoder. The server determines the required data for a client and streams compressed PVs to the client which performs rendering locally using image warping. PVs are predetermined and stored on the server after being compressed. To provide a complex scene, many PVs have to be stored on the server. The system uses the cylindrical panoramas format. Like [49], user navigation is limited to PVs stored on the server and no clear prediction mechanism was identified.

A hybrid geometry/image-based rendering approach is proposed in [79]. Objects with low scene priority such as distant objects are rendered using image-based techniques while other important objects are rendered using the traditional geometric rendering. This enables the system to perform better than pure geometry-based systems since image-based representations are lighter and faster to stream and render than 3D model data. The experiments conducted show that the best performance is achieved using pure image-based rendering, followed by the hybrid approach and the slowest is the geometry-based rendering. The hybrid approach offers a tradeoff between IBR and GBR.

The authors in [53] use remote rendering to protect copyrighted 3D models accessed by remote users. The server maintains the whole 3D virtual environment and sends images to remote clients upon demand. The client renders a low polygon version of the scene while the user is interacting with it. When the user is idle, the server is informed and sends a high-resolution image of the scene back to the client.

The authors in [54] present another work on remote interactive virtual environments where the server renders a 3D virtual world and sends images to a client device. The

client renders novel views using image-based rendering. The main idea of this work relies on sending only the difference between consecutive frames, instead of sending the entire frames. The client sends image requests and wait for the server to reply. Each client uses previous views of the environment to predict the next view, using the known camera motion and image-based rendering techniques. The server performs the same prediction, and sends only the difference between the predicted and actual view. The drawback of this approach is that the client may experience long delays waiting for the next image, because it is a client based request mechanism. Another disadvantage is that it uses lossless compression. The authors used 256x256 images (192KB) for their experiments. It would be relevant to compare this approach with a lossy compression scheme, where entire 256x256 images (20KB) are transmitted in order to evaluate the efficiency of the proposed solution.

In the work presented in [55], the authors explore an alternative approach to achieve 3D graphics capability on mobile devices using image-based rendering. Unlike the geometry-based 3D graphics pipeline, the rendering time of image-based rendering depends on the screen resolution of the output images rather than the complexity of the input models. This offers a potential advantage for mobile devices that typically have small display areas. The authors present a client-server framework for mobile devices equipped with IEEE 802.11b based wireless interface. They have shown a framework in which a 3D graphics programs running on a desktop computer may be used to interact with users on mobile devices. This can simplify the process of developing 3D graphics software for thin mobile devices and offer a way to offload part of the 3D rendering task to the server. Basically, a client device requests images from a server, which renders the 3D environment and send back reference images and depth maps. The client uses the reference images to render intermediate views using McMillan's image warping algorithm [31]. Due to the high capacity server, their scheme can render high quality 3D scenes, which cannot be done on the client through traditional geometry-based rendering. In their experiments, the system is able to render novel views at the speed of 5.9 to 6.2

frames per second on a 206MHz StrongArm processor PDA.

In [56, 57], the authors propose a scheduling and buffering mechanism aimed to improve user interaction with the virtual environment and reduce network latencies. They developed a streaming system where a 3D rendering server sends images to mobile clients. A panoramic renderer on the client renders received images. Cylindrical panoramas are employed in the system. The system predicts user's next views by streaming all possible future positions the user may navigate to from his current position. This had lead to using a simple movement scheme to capture all possible movements. The user may only rotate horizontally and by a specific amount of degrees. The user can not look up or down and navigates forward by a constant speed. This means that possible future user movements have been narrowed to 3 possibilities: move forward, rotate left or rotate right (by a known angle). The system pre-streams these possibilities to the client and therefore achieves path prediction by sending all the possibilities in advance. This system also uses a scheduling mechanism to guarantee quality of service. Problems of this work is that it can not be expanded to include advanced user movement schemes found in many applications today. If the user is allowed to move freely; look up and down and rotate by any amount of degrees, move forward, backward, left and right; then predicting and sending all possible next movements is impossible using this technique.

The authors in [52] propose a client-server approach to provide wireless portable devices with virtual environment exploration. A remote rendering server is responsible for rendering the virtual environment and for the streaming of the rendering outputs to a mobile device client. Only a minor image-based rendering task is left to the resource constrained mobile device. The authors proposed buffering and pre-fetching mechanisms that take into consideration the user's movement pattern and that adapt the rate and schedules the images that will be pre-fetched.

2.6 Comparative Analysis

In this section we provide a comparison between the multitude of virtual environment remote rendering approaches and techniques we have discussed in this chapter. Table 2.1 contrasts between image-based and geometry-based rendering on mobile devices.

GBR renders geometry on the client and therefore complex scenes cannot be supported and only low polygon, low quality textured models can be displayed. It consumes a lot of bandwidth to stream the 3D models. Techniques discussed in section 2.4 are used to minimize the network load. GBR provides the remote user with more interactivity in terms of user navigation freedom and scene changes where IBR usually delivers static scenes to the client.

IBR is lighter to render on the client since rendering images is much faster and lighter than rendering geometry. Therefore, complex environments can be rendered with excellent resolutions since IBR does not depend on the scene's complexity, it rather depends on the image resolution. IBR consumes less bandwidth as well.

Table 2.1: **IBR vs GBR on mobile devices**

Technique	*Rendering Load*	*Complex Scene Support*	*Interaction*	*Network Load*	*Display Quality*
GBR	High	Low	Complex	High	Low Polygon
IBR	Low	High	Simple	Moderate	High Resolution

Table 2.2 compares between the image-based remote rendering approaches we discussed in the previous section and our proposed streaming protocol; *Partial Panorama Prediction*. The table includes the following criteria:

- **Navigation:** The navigation options the user has. It can be limited to fixed paths, or certain movements or be free.
- **Prediction:** Indicates if the streaming approach actually predicts the user's next movements, by studying his movement patterns or using probabilities, etc...
- **Caching:** Indicates if scene data that might be used is stored on the client for improved performance.

- **Adaptivity:** Shows if the streaming system changes its behavior according to dynamic events such as user movements.
- **Performance:** How fast the streaming system delivers the requested scenes to the mobile client.
- **Network Load:** The amount of network bandwidth consumed.
- **Display Problems:** Problems might result in the final images the user sees due to IBR algorithms such as distortions, occlusions and lost pixels.

It is noted that user movement prediction is absent in almost all of the existing systems except for the ones that achieve it by restricting user movement to either static paths [51], or limited movement options [56, 57, 52]. Our new proposed system predicts the user's next movements and grants the user an advanced motion scheme at the same time. It predicts future movements by caching data on the client necessary to generate new views at the newly predicted positions. The works that rely on panoramas; cache rotations on the client since a panorama covers rotations at a certain position [47, 48, 51, 49, 80, 56, 57, 52]. The works that use predefined set of paths [51, 49, 80] or a limited movement scheme [56, 57, 52] cache user movements to these predefined positions. Works like [55, 48] which use image warping or view morphing, can generate views at arbitrarily close positions but their generated images suffer from problems such as distortions and dependency on reference image in case of view morphing, lost pixels, occlusions and holes in image warping. None of the systems we found adapts its self to events such as user movement patterns; they only *react* after being informed of what the user requests. Our new idea is *proactive*; changing its behavior and optimizing its prediction according to the user's navigation manner. In terms of network load, panorama based systems usually consumes a lot of bandwidth since panorama sizes are large. Systems that accompany using panoramas with algorithms such as view morphing to generate images at intermediate positions have a slighter advantage. Our new idea

sends important information in advance to boost the performance which results in an above average network consumption. We try to minimize the bandwidth consumption by sending *partial panoramas*; panoramas covering a limited range of rotations, which will be explained later in this thesis and by compressing depth values that are used to reconstruct new views at the client. We face the problem of occlusion which is caused by revealing objects at the new position which were hidden in the original position. There are many existing solutions to this problem in the literature [34]. We currently consider this problem to be out of the scope of this thesis and leave it for future work.

2.7 Summary

In this chapter we discussed multimedia streaming in general. We identified the special issues that arise from using wireless networks and differentiated between interactive and non-interactive media. We showed various approaches used to stream 3D virtual environments across networks. We described geometry replication, progressive meshes, impostors and image-based rendering as alternatives to stream virtual environments on remote mobile clients. Image-based rendering has many techniques in the literature, some does not use any geometry for rendering such as lightfield and lumigraph, others use implicit geometry for rendering such as view morphing and view interpolation and other use explicit 3D geometry such as McMillan's warping algorithm. All IBR techniques are based on the plenoptic model and implement at least a subset of it. Panoramas capture all light rays from a certain point in the virtual environment and are usually projected on a 3D shape such as cubes, spheres and cylinders. We listed existing solutions that try to stream virtual environments to mobile devices, compared between them and showed their strengths and weaknesses. In the next chapter, we will explain the virtual environment streaming system we developed for this thesis. This system provides infrastructure and facilities such as network access, session management and message serialization that will be used by the VE streaming protocols that we develop in this thesis.

Table 2.2: **Image-based remote rendering approaches comparison**

Technique	Literature	Navigation	Prediction	Caching	Adaptivity	Performance	NW Load	Display Problems
Chromium Based	[78]	Free	Absent	None	Absent	Average	Average	None
Panorama	[47]	Fixed	Absent	Rotation	Absent	Slow	High	None
Panorama + View Morphing	[48]	Free	Absent	Rotation, close translation	Absent	Average	Above Average	Distortion
Fixed Panorama Pre-fetching	[51]	Fixed	Primitive	Rotation, fixed translation	Absent	Above average	High	None
Panoramic Video (PV)	[49, 80]	Fixed	Absent	Rotation, fixed translation	Absent	Below Average	High	None
Hybrid Image/Geometry-based	[79]	Free	Absent	None	Absent	Below Average	Above average	None
Image warping	[55, 80]	Free	Absent	Rotation, close translation	Absent	Above average	Average	Occlusion, holes, missing pixels
Simple Panoramic Prediction	[56, 57, 52]	Limited	Adequate	Rotation, fixed translation	Absent	Above average	High	None
Proposed Partial Panorama Prediction	This thesis	Free	Advanced	Rotation, translation	Adaptive	Fast	Above average	Occlusion

Chapter 3

Architecture and Design of the Virtual Environment Streaming System

The first step in tackling the problem of displaying complex virtual environments on a mobile device is designing a system which defines the roles of the elements involved and the relations between these elements, and provides a unified infrastructure and services necessary for the yet to design streaming protocols. In this chapter, we describe the selected architecture which is client-server. The server maintains the virtual environment and the mobile clients communicate with the server to access the virtual environment. We then specify the subsystems within each of the server and the mobile clients in detail and explain the common messaging system used for communication. We then explain the abstract design of the protocol components and how it fits into the system.

3.1 Virtual Environment Streaming System Architecture

The system architecture is client-server. Since the clients are light and do not possess powerful processing or 3D rendering capabilities, the virtual environment including its graphics and states will reside on a server which is a powerful computer with powerful graphics accelerator capable of rendering and maintaining complex 3D worlds.

Clients will communicate with the server via a wireless network. Actual network devices and mediums depend on the application of the system. Clients will send requests and notifications to the server which in turn, reacts and feeds the clients with the data needed to display the environment. A user database will be used to validate client credentials upon signing in.

The protocols that will be designed in this thesis all depend on Image-Based Rendering. In this approach, the server renders a view and sends an image to the client which displays the image. Figure 3.1 shows the high level architecture of the developed streaming system.

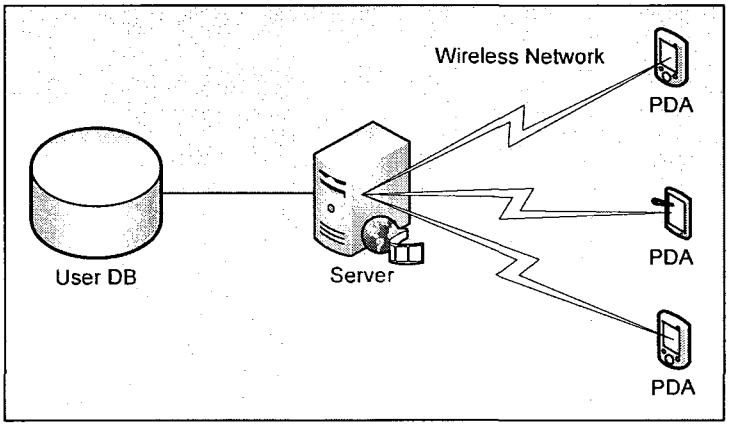


Figure 3.1: Virtual Environment Streaming System Architecture.

To achieve the goals of streaming views to multiple connected mobile clients while preserving system maintainability, efficiency and modularity, both the server and the

client softwares have to be designed carefully and split into reusable, clear components that collaborate together to achieve these goals. In the following two sections, we show the anatomy of both the server and the client in detail and show how these sub systems operate together to display the virtual environment on the client device.

3.2 Server Components Design

Figure 3.2 shows the server application main components.

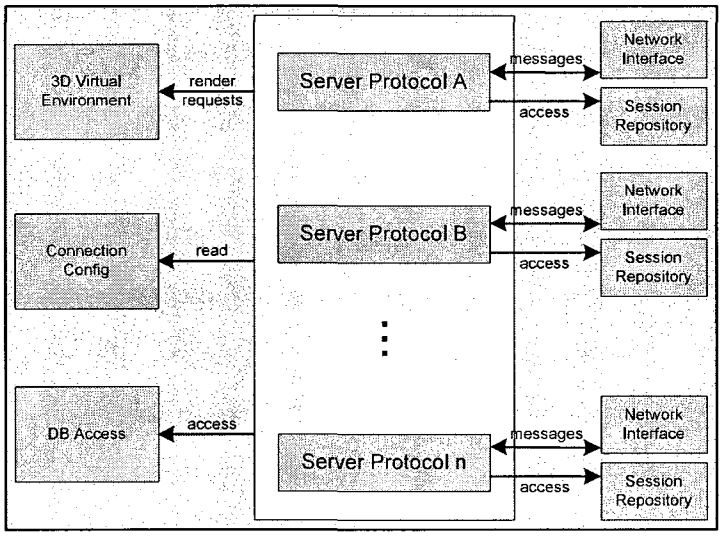


Figure 3.2: Server Application Main Components.

- **3D Environment**

  This component contains the virtual environment model and is responsible for rendering and updating the environment. This component will be accessed from the various server streaming protocols to provide necessary imagery needed to fulfill client requests.

- **Streaming Protocols**

  The server contains various streaming protocols responsible for handling client re-

quests and notifications. Each protocol is configured to listen on a different network port. This enables running all the protocols simultaneously to handle different incoming requests from clients implementing different corresponding protocols. Apart from accessing the virtual environment, streaming protocols need the below listed support to function successfully.

- **Network Interface**

  This component isolates network protocol implementation and enables the server protocol to seamlessly send messages to clients and notifies it with incoming client messages.

- **Session Repository**

  This component stores valuable information about each client, such as ID, world position and orientation (where the client is facing), movement history, status and other protocol specific data.

- **Protocol Network Configuration**

  This component configures each protocol to listen to a specific network port.

- **Database Access**

  Protocols need to validate user names and passwords and access some stored information about the virtual environment. This component facilitates database access for the protocols.

When the server starts up, it initializes the 3D environment and creates the streaming protocols which start listening to incoming messages.

3.3 Mobile Client Components Design

Figure 3.3 shows the client application components.

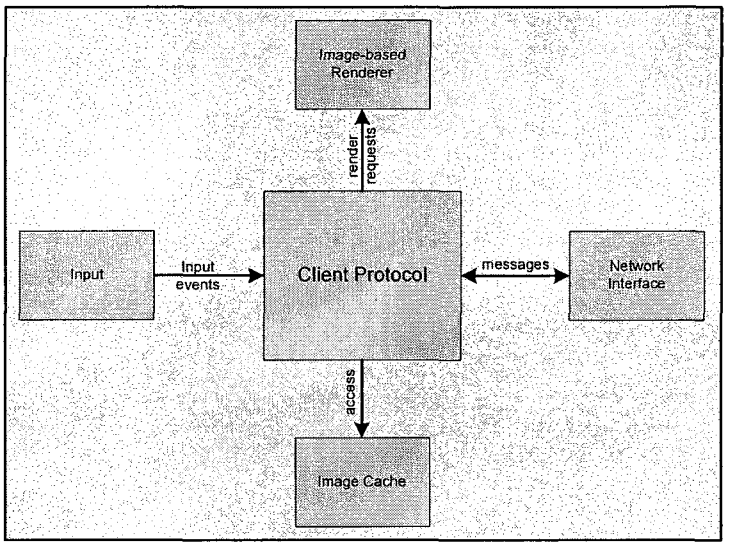


Figure 3.3: Client Application Main Components.

- **User Input**

  Reads user inputs such as key presses, pointing device moves and clicks and notifies the client streaming protocol to act accordingly.

- **Client Streaming Protocol**

  This component implements the client side of the streaming protocol. It sends requests to the server, receives image data from the server and in some protocols, stores them in a cache, and sends rendering commands to the image based renderer.

- **Image-Based Renderer**

  Depending on the protocol, the image-based renderer receives image data in a certain format and renders the view on the client device screen.

- **Image Data Cache**

  Used by some protocols to store image data that most probably will be needed in the near future.

- **Network Interface**

  This component is responsible for sending messages over the network to the server and notifying the client protocol of incoming server messages.

This was a high level description of the components of the streaming server and the mobile client. The details of how these components actually work and synchronize, depend on the streaming protocol and will be discussed in later chapters where we describe the streaming protocols that have been developed in this thesis. In the next section, we will show how the communication between the server and the client is done and explore how the messages are formatted and exchanged between them.

3.4 Network Communication and Messaging

The server and the client use a network to exchange messages containing requests, notifications and scene data. We developed a common network interface layer on both the server and the client, which has the following characteristics:

1. Provides a standard format for messages exchanged between the server and the client.

2. Abstracts the actual network protocol used to send the messages.

3. Establishes and closes network connections between the server and its clients.

4. Provides automatic serialization of messages into byte streams to be sent over the network and automatic deserialization of received data into the standard message format mentioned above.

5. Incoming messages notification. The calling code can register to handle specific incoming messages.

3.4.1 Standard Messages Format

There are two different message formats, one for server messages and another for client messages. Figure 3.4 shows the UML class diagram for the server message.

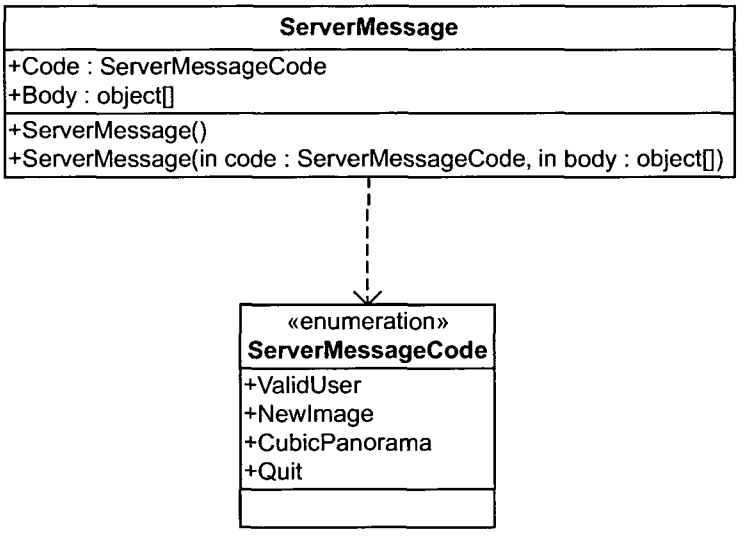


Figure 3.4: *ServerMessage* and *ServerMessageCode* Class Diagram.

A server message has a code which specifies the type of the message. For example, a *NewImage* message contains the data for an image to be displayed at the client and a *ValidUser* message indicates that the client has successfully signed in. The above diagram only shows a subset of server message codes. Later on when we discuss the protocols, we will show all the different message types used by the server. The server message also has a body that holds the actual data of the message. The body is implemented as an array of objects. This enables sending multiple parameters of different data types to the client. Below is an example of configuring a new image server message.

Listing 3.1: Configuring a server image to hold image data

```
// create a new server message
ServerMessage message = new ServerMessage();
// set its code to new image
message.Code = ServerMessageCode.NewImage;
```

```
// this message has 2 parameters
message.Body = new object [2];
// first parameter is the image data
message.Body [0] = imageToBeSent;
// second parameter is the image size
message.Body [1] = imageToBeSent.Size;
```

Client messages have a similar structure. They include an additional User Id field that identifies the client.

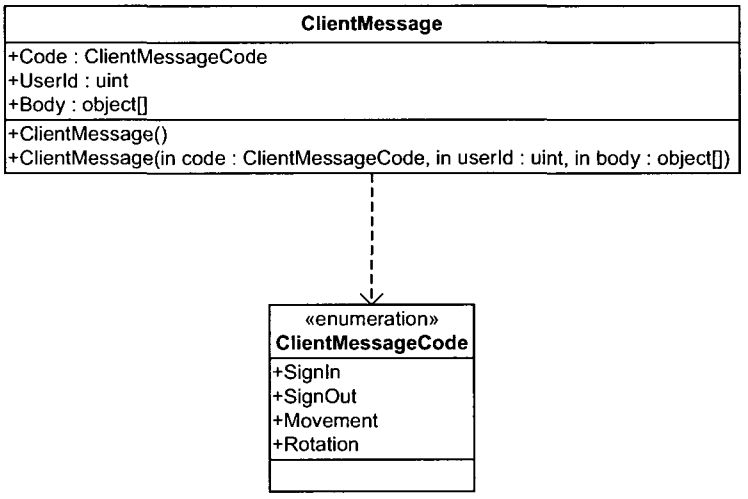


Figure 3.5: *ClientMessage* and *ClientMessageCode* Class Diagram.

Client message codes included here are only partial. Examples include signing in and out of the server and informing the server of a movement or a rotation. Here is an example of configuring a message to be sent to the server upon a forward movement on the client.

Listing 3.2: Configuring a client message to store a forward movement

```
// create a new client message
ClientMessage message = new ClientMessage();
// set the user id of the client
message.UserId = 13;
// set the code of the message to movement
```

```
message.Code = ClientMessageCode.Movement;
// this message has one parameter
message.Body = new object [1];
// inform the server that the client has moved FORWARD
message.Body [0] = Movement.Forward;
```

3.4.2 Network Protocol Abstraction

Having the messages format specified, the next task is exchanging them over the network. Protocols such as TCP or UDP can be used for this task. We designed our network communication modules to be independent of the network protocol used. The communication module is associated with a generic network protocol from which specific implementations such as TCP or UDP can be derived. Figures 3.6 and 3.7 show UML class diagrams for the network protocols on the server and the client respectively.

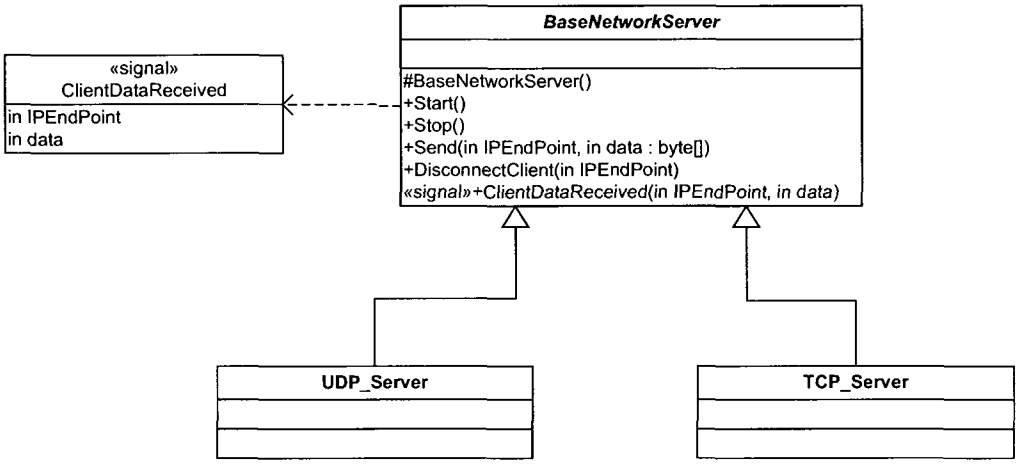


Figure 3.6: Server network protocol abstraction classes.

BaseNetworkServer is an abstract network server class. It provides operations common to any network server application such as listening to a server port, disconnecting a client, sending a message to a client and notification of a received message. From this class, different implementations can be derived such as TCP or UDP. These implemen-

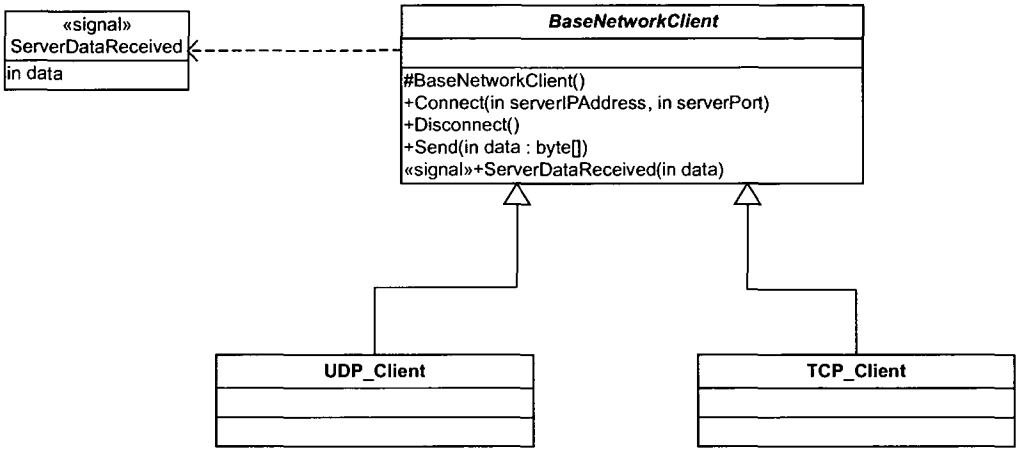


Figure 3.7: Client network protocol abstraction classes.

tations share the same common interface; *BaseNetworkServer* . Therefore the actual implementation of the network protocol is now isolated.

A similar approach was taken on the client. *BaseNetworkClient* encapsulates common network client behavior, connecting and disconnecting to a server, sending messages and notification of receiving a message from the server. Two implementations of the network client protocols are provided; TCP client and UDP client.

3.4.3 Network Communication Modules

The final step in completing the network communication module is providing an interface that hides all the network details from the calling modules.

The model in Figure 3.8 shows the complete picture of the server network communication module. *ServerCommunicationAgent* class is the interface for the network communication module on the server. It uses an implementation of *BaseNetworkServer* internally to accomplish the job. Clients of this class can access network services easily and seamlessly by calling the *Start()* method which starts network communication, then can subscribe to be notified for certain incoming messages from clients by calling the *HandleEvent()* method and passing it the client message code and a callback function

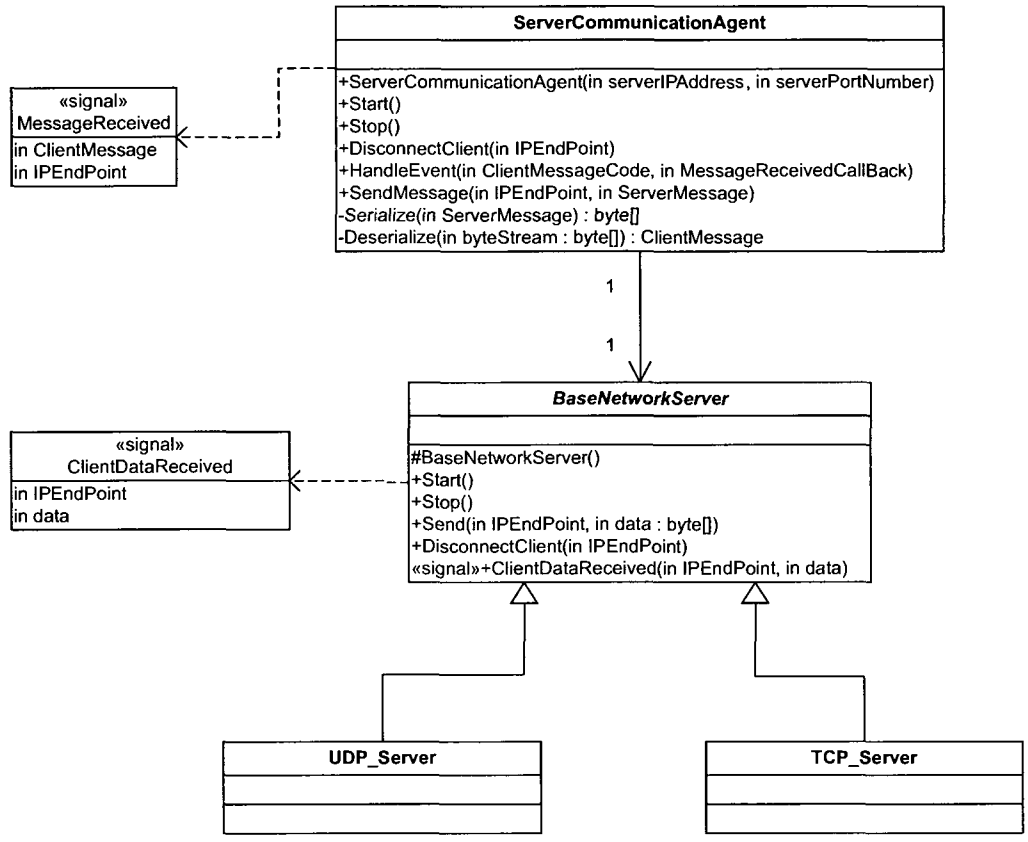


Figure 3.8: Server network communication module.

to call upon receiving the message. When the callback function is called, it receives the network address of the client and a client message that is automatically deserialized. Users of this class can send messages in a high level manner by passing a server message and a network address, serializing the message and sending it using a network protocol is taken care of internally. The *Stop()* method shuts down the network communication module. The client has a similar design which is shown in Figure 3.9.

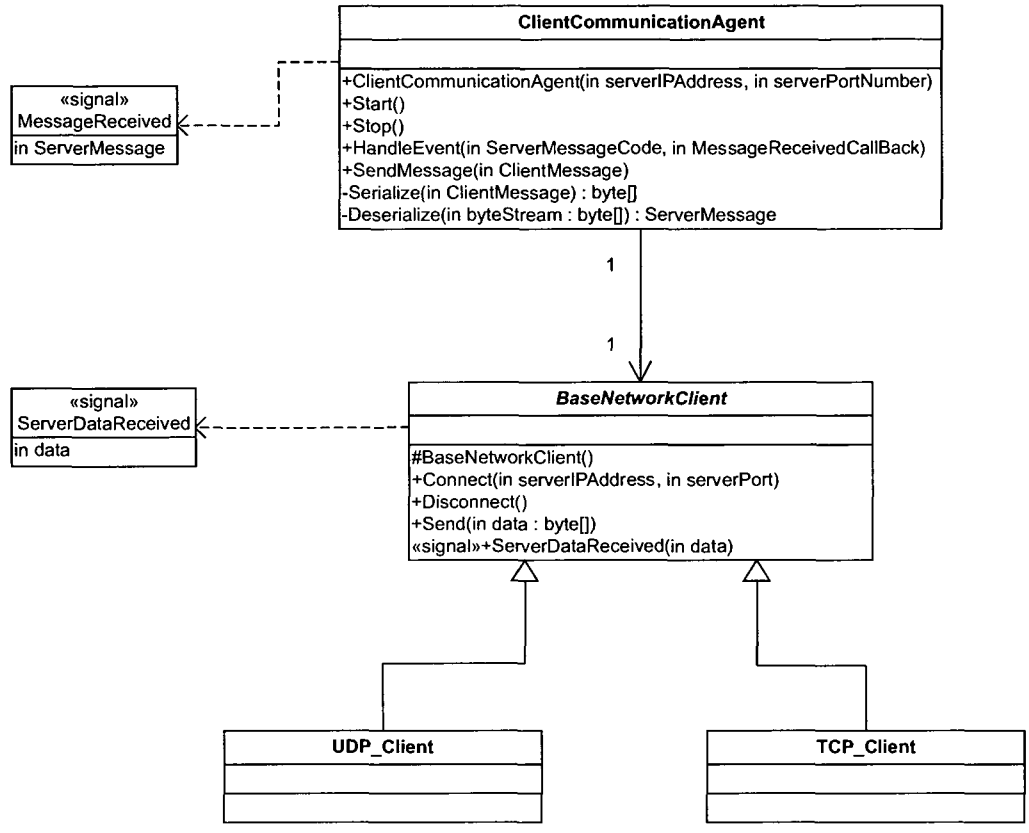


Figure 3.9: Client network communication module.

Here is a working example of a client signing in to the server and the server sending back the appropriate response.

Listing 3.3: Client sign in

```
// create a client network agent and set it to connect to a server
// on the local host on port 999
ClientCommunicationAgent client = new ClientCommunicationAgent("127.0.0.1", 999);
// handle the valid user and error incoming messages from the server
client.HandleEvent(ServerMessageCode.Error, onErrorMessage);
client.HandleEvent(ServerMessageCode.ValidUser, onValidUserMessage);
// start network communication
client.Start();
// send a sign in message to the server
```

```
ClientMessage signInMsg = new ClientMessage();
signInMsg.Code = ClientMessageCode.SignIn;
signInMsg.Body = new object [] {username, password};
client.SendMessage (signInMsg);

. . .

// this function will be called when the server sends a valid user message
public void onValidUserMessage(ServerMessage message)
{
    // show a success message
    MessageBox.Show ("Sign in successful!");
}

// this function will be called when the server sends an error message
public void onErrorMessage(ServerMessage message)
{
    // show the server error message
    MessageBox.Show ("Sign in Failed. Server Error Message:" + message.body [0]);
}

public void onApplicationExit()
{
    // when the client application closes, stop network agent
    client.Stop ();
}
```

Listing 3.4: Server sign in

```
// create a server network agent and set it to listen on port 999
ServerCommunicationAgent server = new ServerCommunicationAgent ("127.0.0.1", 999);
// handle sign in messages incoming from the client
server.HandleEvent (ClientMessageCode.SignIn, onClientSignIn);
// start network communication
server.Start ();
```

```
. . .

// this function is called when the server receives a sign in message from a client
public void onClientSignIn (ClientMessage message, IPEndPoint address)
{
    // read user name and password from the client message
    string username = (string) message.Body [0];
    string password = (string) message.Body [1];

    // Do what ever to sign in the client
    . . .

    ServerMessage response = new ServerMessage ();

    if (signedIn) // user signed in successfully
    {
        response.Code = ServerMessageCode.ValidUser;
    }
    else
    {
        response.Code = ServerMessageCode.Error;
        response.Body [0] = "invalid user name or password";
    }

    // send the response message to the client
    server.SendMessage (address, response);
}
```

3.5 Server and Client Protocol Design

Having the network, virtual environment and all other sub systems developed and ready, we only need a component where we can implement the different client/server streaming protocols. In this section, we describe the server and the client protocol design and how it uses all the other aforementioned components and services to achieve our goal of displaying virtual environments on a mobile device.

3.5.1 Server Protocol Design

The server protocol is the most important component on the server; it represents the server logic of the streaming protocol and uses other components such as networking, session management and virtual environment to accomplish its job. Figure 3.10 shows the structure of the server protocols and how they relate to other components.

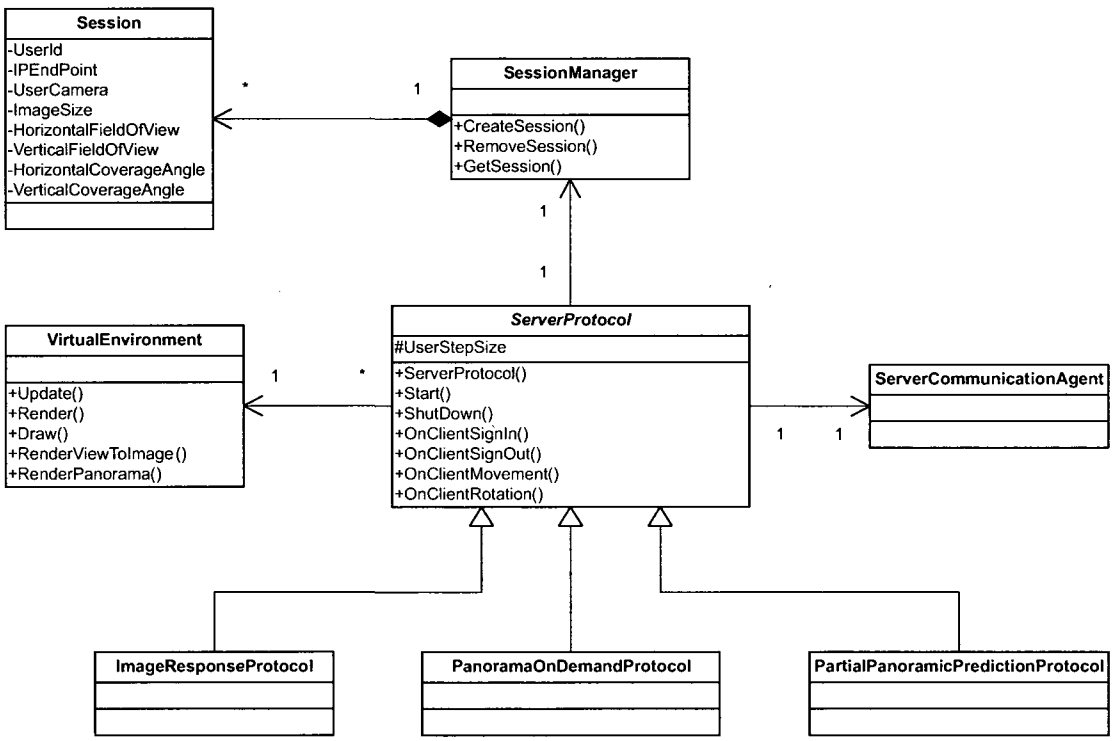


Figure 3.10: Server protocol design.

A generic server protocol class *ServerProtocol* holds the common attributes and behavior of a server protocol. A server protocol can be started and shut down and has associations with other subsystems. It uses *ServerCommunicationAgent* class discussed in section 3.4 to access the network and communicate with clients. It also has access to the virtual environment and sends rendering requests depending on the specific implementation of the protocol. It uses the session management component to store and read clients important information. As the figure shows, a client session includes its ID, IP address, camera in the virtual world and other protocol specific data which will be described in later chapters. Equipped with all these facilities, we can implement the server side of a streaming protocol with ease; we only concentrate on the logic of the protocol since the rest of the work is already implemented. There are three protocol implementations shown in the diagram which will be explained in later chapters.

3.5.2 Client Protocol Design

The client follows the same design as the server. The *ClientProtocol* class encapsulates common client protocol behavior. It associates with *ClientCommunicationAgent* to communicate with the server and displays received imagery to a standard windows forms control. Some implementations such as those involving panoramas need a simple virtual environment to render panoramas and Other implementations cache received data for future use. Client protocol design is depicted in Figure 3.11.

3.6 Summary

In this chapter, we described the virtual environment streaming system that will be used as an infrastructure for the streaming protocols that will be described in the next two chapters. We employed client/server architecture and described the components interacting on each side to lay the ground for streaming protocols. We have shown how we handled the network communication and how the protocols are to be fitted into the

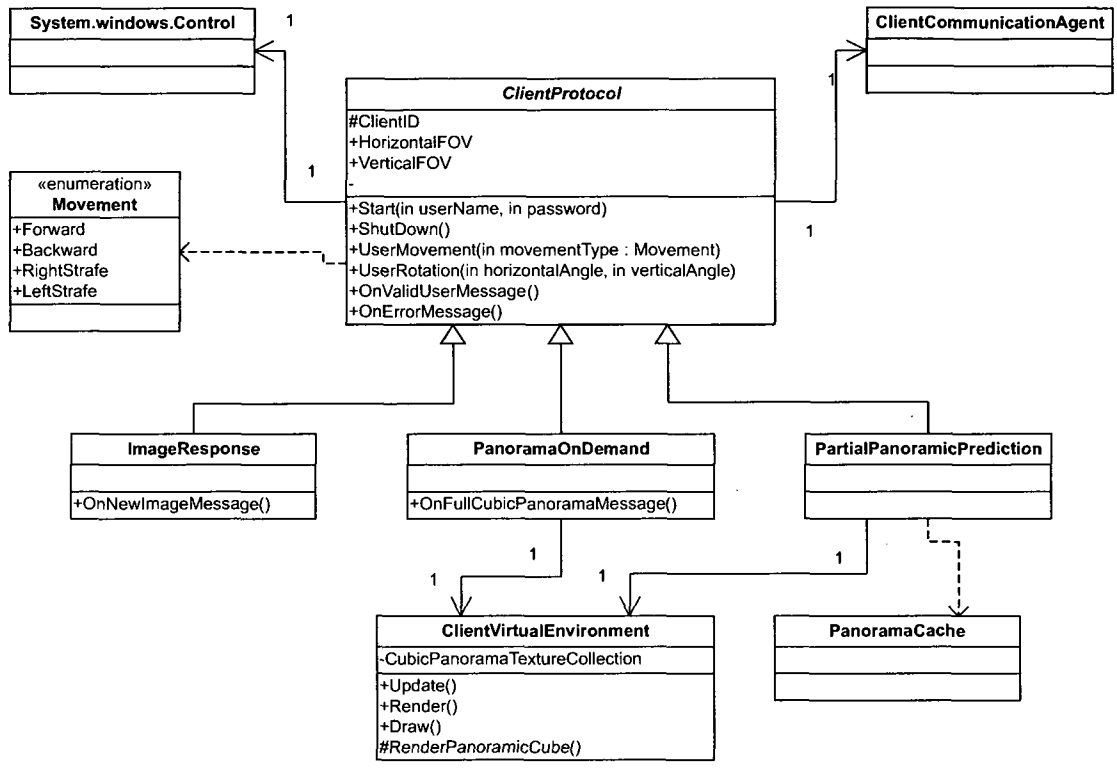


Figure 3.11: Client protocol design

system. It is now time to talk about the different streaming protocols we developed and discuss them in details.

In the next chapter, we introduce two streaming protocols, *image response (IR)* and *panorama on demand (POD)*. We shall describe their ideas and analyze them from many points of view, such as efficiency, movement prediction, bandwidth utilization and many more.

Chapter 4

Image Response and Panorama on Demand Streaming Protocols

In this chapter, we introduce two image-based streaming protocols. The first, *image response (IR)*, is the most straightforward. The server sends images of new views based on client movements. The second protocol, *panorama on demand (POD)*, uses some sort of path prediction where for any position in the virtual environment, the server sends a full panorama to the client. The client then can use the panorama to provide all views at the current location.

Most of the existing distributed virtual environment systems discussed in section 2.5 can be classified into one of these protocols. Therefore these two protocols performance can approximately measure these systems performance.

In this chapter we describe the ideas and motivation behind these two protocols. Then we describe our implementation of these protocols. Finally, we analyze the protocols and discover their advantages and shortcomings and set the motivations for innovating a new idea that overcomes these limitations.

4.1 Image Response Protocol (IR)

When using image-based rendering (IBR) to stream virtual environments to mobile devices, first thing that comes to mind is rendering the virtual environment at the current location of the user, saving the scene to an image and sending it to the client. The client simply receives the image and displays it. This is the basic idea of the IR protocol.

Simplest does not always mean best. In this approach, upon every client movement, the server *reacts* and sends new images. The server does not do any job to provide the client with images in advance since it does not know what could happen, it only knows *after* it happens. Therefore this idea is simple but completely lacks prediction.

The client is completely bound to a network and server processing delay before it can provide the user with the new view. As the results will show later, this protocol consumes the least network bandwidth but in return it does not provide the user with highly interactive responses.

In the next sub section, we describe our implementation details of the IR streaming protocol.

4.1.1 IR protocol Implementation

Using the system we described in chapter 3, the IR streaming protocol is implemented by inheriting *ServerProtocol* and *ClientProtocol* classes for both server and client side protocols. The server listens on a special port used to receive IR client protocol requests. Figure 4.1 shows the sequence of actions that take place when a client signs in to the server.

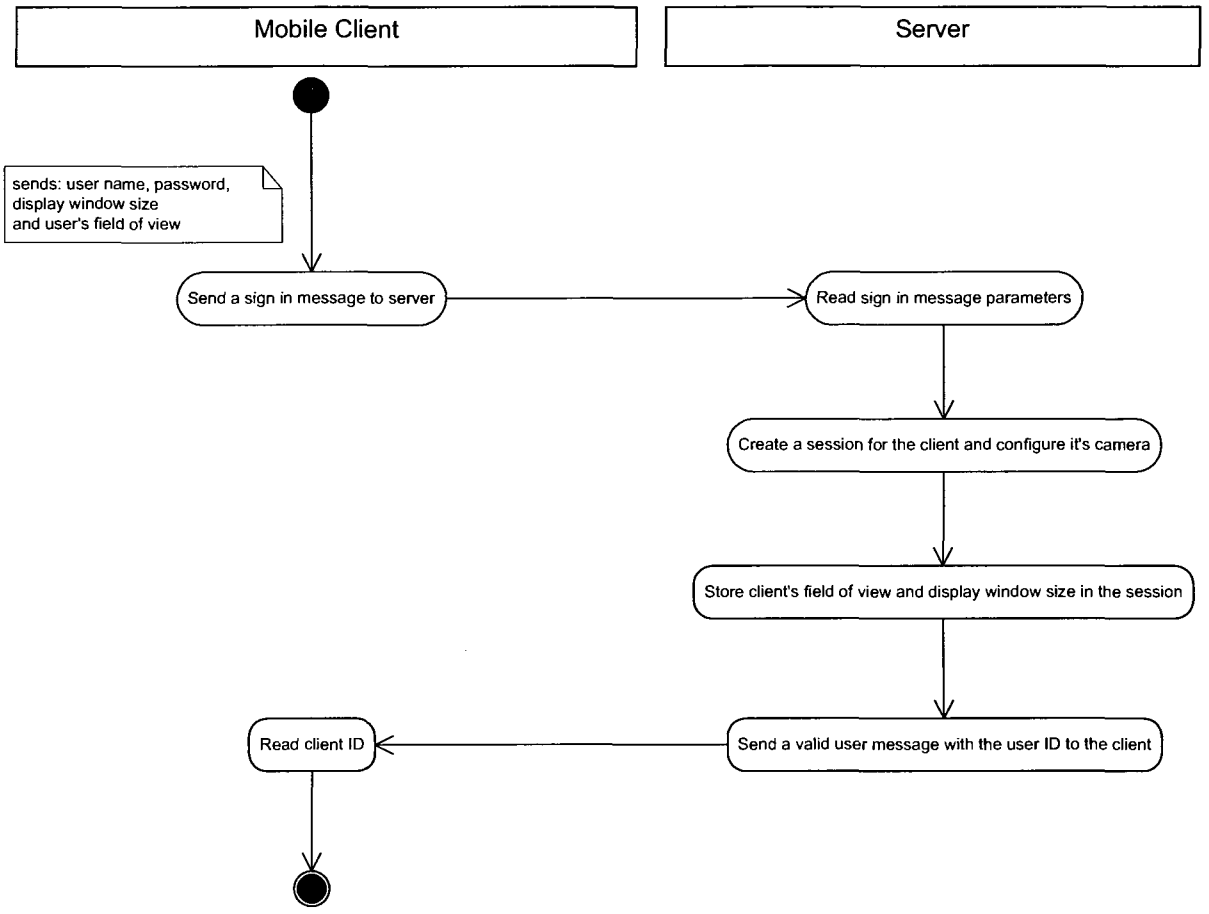


Figure 4.1: IR client sign in activity diagram.

The mobile client sends a *SignIn* message to the server with the following parameters:

- **User name.**
- **Password.**
- **Display Window Size:** dimensions of the window used to display the image. Determines the image size.
- **Field of View:** client's horizontal and vertical field of view.

The server receives the sign in message and reads its parameters. It checks user's

credentials and creates a new session for that user. It then configures the camera, places it in the virtual environment and applies the client's vertical and horizontal field of view (FOV). Client window size is stored in the session as well. Finally, the server sends a *ValidUser* message to the client attached with the user's ID.

Once the protocols start up, the client is now able to navigate in the virtual environment. On each movement issued by the client's input module, the client protocol is notified with the movement. Figure 4.2 shows the flowchart of actions that taking place when a client moves or rotates view in the virtual environment.

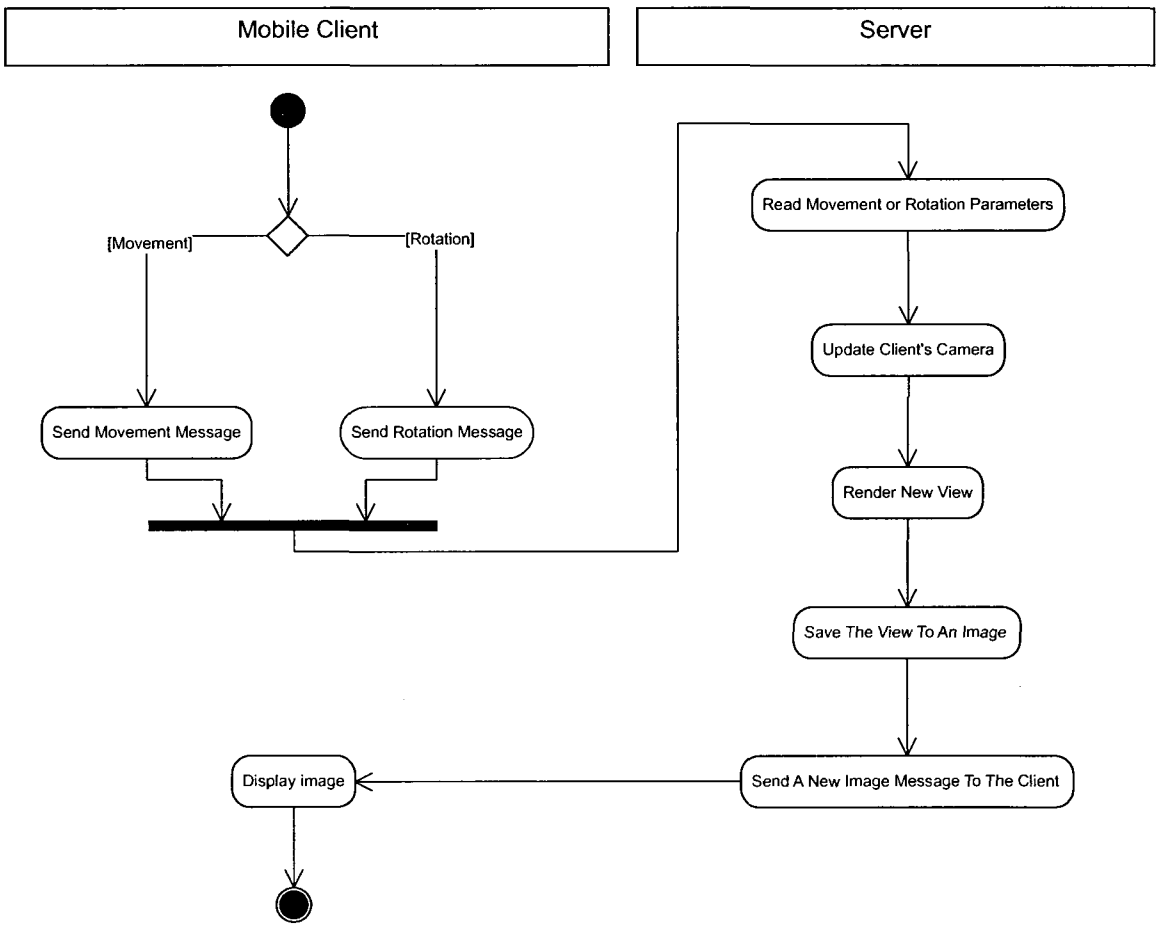


Figure 4.2: IR movement/rotation activity diagram.

The client sends a *Movement* or a *Rotation* message depending on the movement. The server updates the client's camera, requests the virtual environment to render the new view, then saves the new scene to an image. To save bandwidth, the image is encoded as a PNG image. PNG was selected since it is a *lossless* compression algorithm; exact pixel values can be restored after decoding the image. This feature will prove to be very useful in later protocols. Finally, the image is sent to the client who displays it to the user.

4.1.2 IR protocol Analysis

Following are the advantages of the IR protocol:

1. Simplicity. It is the most straightforward method to implement image-based streaming.

2. Consumes the least bandwidth per movement since it does not send additional information that predicts next movements.

Shortcomings of the IR protocol are listed below:

1. Does not use any prediction mechanism to make next predicted views available on the client in advance.

2. Response time always depends on the network delay. On heavy network load, user experience may degrade due to increased network latency and the lack of availability of images on the client. Other protocols may rely on local images on the client to improve system responsiveness in case of network overload.

4.2 Panorama on Demand Protocol (POD)

The POD protocol tries to solve the lack of prediction problem in the IR protocol. Whenever the client changes position in the virtual environment, a panorama that covers

all rotations at the new position is sent to the client; thereby localizing rotations at the new position and eliminating the need for transmitting new images for these rotations.

This idea works best if it is well known that the user will rotate a lot at every position her/she visits. The larger the number of rotations the better the protocol performs. A big problem arises if the user changes position a lot; this leads to performance much worse than IR since the size of a panorama is considerably larger than a simple image.

4.2.1 POD Protocol Implementation

In our implementation of the POD protocol, we used cubic panoramas. For more information about panoramas and their different types please refer to section 2.2.4. POD protocol is implemented in the standard way of inheriting *ServerProtocol* and *ClientProtocol* classes.

When a POD client signs in, the same sequence of actions in Figure 4.1 executes. The difference is in handling movements and rotations. In this protocol, the client maintains a simple virtual environment that displays a cube. The user is positioned at the center of the cube. Look around (rotations) is achieved by rotating the client camera. Movement is done by replacing images that the cube faces display. The user camera never actually moves. The cube panorama is made of six images, one image for each face. Each image is applied to a cube face as a texture. The client virtual environment is rather simple, since it only uses a few vertices for the cube and six textures and therefore can be loaded easily on the mobile device. Figure 4.3 shows the actions that undergo when a client rotates. As noticed from the flowchart, the rotation is handled locally on the client. Therefore, in a case of rotation, the POD protocol is much faster than IR. However, when the client moves, a whole new panorama has to be rendered on the server and streamed back to the client, as shown in figure 4.4. Since the client rotations are not reported to the server, upon moving, the client has to inform the server where the user is facing i.e. Horizontal and vertical rotations. The server updates the client camera and renders a new cubic panorama.

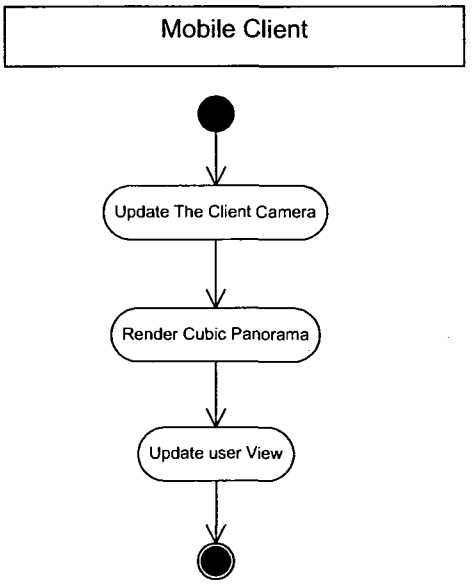


Figure 4.3: POD rotation activity diagram.

Let's explain how the server renders the cubic panorama. A cubic panorama captures all the views the user can look at from a certain position. It captures six orthogonal images corresponding to the cube faces. The camera's field of view is set to 90 degrees both horizontally and vertically. The front cube face is rendered first. Then the camera is rotated horizontally by 90 degrees to render the right face. Same thing goes for rendering left and back cube faces. The upper cube face us rendered is rotating the camera vertically by 90 degrees and the lower cube face is rendered by rotating the camera by 270 degrees vertically.

The server maintains a buffer to which scenes are rendered. This buffer is called the *back buffer*. The back buffer has a maximum image size which the server can render to. Since a cubic face image represents a $90° \times 90°$ field of view, its size depends on the client's field of view and image size. For example if the user's field of view is $45° \times 45°$ and its window size is 100×100. Simply rendering 100×100 cube faces will result in distorted and low quality images seen by the client since this small image have to be stretched on the client to cover a $90° \times 90°$ field of view. To maintain image quality, the

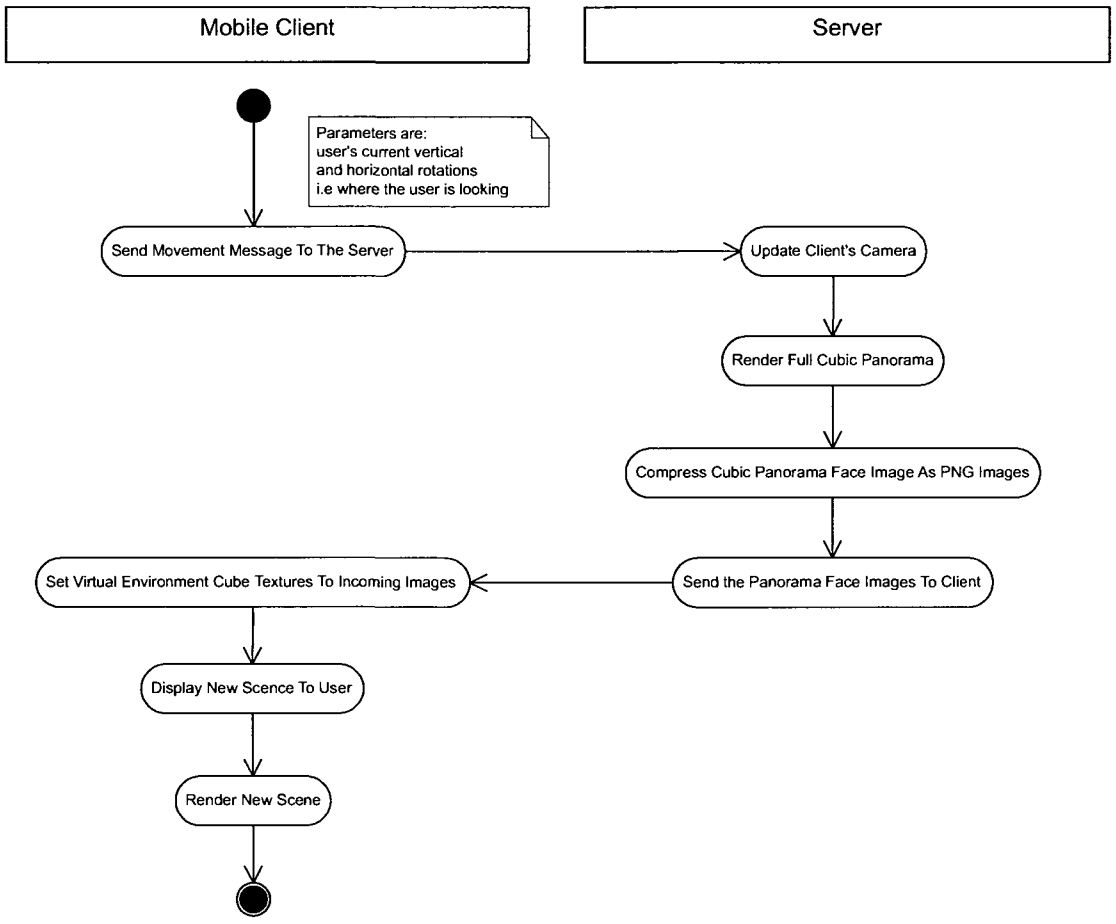


Figure 4.4: POD movement activity diagram.

size of a cube face image is computed by the following formulas:

$$\begin{aligned} Height &= \frac{90}{clientVFOV} \times clientWindowSizeHeight \\ Width &= \frac{90}{clientHFOV} \times clientWindowSizeWidth \end{aligned} \tag{4.1}$$

This means that the cube image size relative to the client's window size is:

$$(90/clientHFOV) * (90/clientVFOV) \tag{4.2}$$

So if the client's field of view is $45° \times 45°$ then the size of a panorama face image will be 4 times the size of the client window size in order to maintain image quality.

After computing the cube face image size, the server renders to a subset of the back buffer equal to the size of the cube face. This is done by setting a *view port* on the back buffer. The following code snippet shows the routine that renders a cubic panorama for a client's position:

Listing 4.1: Cubic Panorama Rendering

```
// Renders a full 360 cubic panorama from a camera's view
// clientCamera: Client's camera
// hFov: Client's horizontal field of view
// vFov: Client's  vertical field of view
// windowSize: Client's window size
// returns: generated panorama
public CubicPanorama RenderFullCubicPanorama(Camera clientCamera,
                        Degree hFov, Degree vFov, Size windowSize)
{
    // create a view port with the size of a cube face
    Viewport newViewport = new Viewport();
    newViewport.X = newViewport.Y = 0;
    // width and height of the view port depend on the client's field of view
    // and window size
    newViewport.Width = (int)((90.0 / hFov) * windowSize.Width);
    newViewport.Height = (int)((90.0 / vFov) * windowSize.Height);
    // set projection matrix to 90 degrees both vertical and horizontal
    ProjectionMatrix = Matrix.PerspectiveFovLH(90, 90);
    // Assign the new port to rendering device
    Device.Viewport = newViewport;

    CubicPanorama cubicPanorama = new CubicPanorama();
    Bitmap bitmap;

    // start rendering cube faces...

    // Front Face
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Front] = BackBuffer.GetPNG();

    // Right Face
    clientCamera.RotateRight(90);
```

```
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Right] = BackBuffer.GetPNG();

    // Back Face
    clientCamera.RotateRight(90);
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Back] = BackBuffer.GetPNG();

    // Left Face
    clientCamera.RotateRight(90);
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Left] = BackBuffer.GetPNG();

    // reset the camera to face original direction
    clientCamera.RotateRight(90);

    // Up Face
    clientCamera.RotateDown(Geometry.DegreeToRadian(-90));
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Up] = BackBuffer.GetPNG();

    // Down Face
    clientCamera.RotateDown(Geometry.DegreeToRadian(180));
    Render(clientCamera);
    cubicPanorama[CubicPanoramaFace.Down] = BackBuffer.GetPNG();

    // return the cubic panorama
    return cubicPanorama;
}
```

Now that the cubic panorama is rendered and compressed, the server sends it to the client via the networking foundation discussed in chapter 3. Once it receives a new panorama message, the client applies the new panorama face images to its virtual environment cube; assigning each cube face the appropriate image. It then renders the textured cube and the user is shown the new view at the new position.

4.2.2 POD Protocol Analysis

In summary of the POD protocol, it uses a simple form of predicting future user movements. It makes images needed for future rotations available on the client in advance but does not take into account user movements. As discussed above, the size of a panorama face image is usually much bigger than the image the client sees. For the case of $45° \times 45°$ field of view, it is 4 times the client window size. This yields to sending 24 times the images for a whole panorama since it has 6 faces. This size is much bigger and therefore much slower to transmit. It can be seen that if the user changes position a lot, this protocol will perform poorly. Most of the time; the client will only use a small amount of the panorama, if it rotated at all. Overall and as the experiments will show; IR is simpler and far better than POD.

4.3 Summary

In this chapter, we discussed two streaming protocols; IR and POD. We described how each of them works, showed how we implemented both in our streaming system, and discovered their strengths and weaknesses.

These protocols suffer from being action driven i.e. they react once a user signals a request. The user is always ahead of the system, issuing requests and the system responds. What is needed is new idea that predicts almost all possible future user movements and pre-sends only information that will be used by a next movement i.e. not send unnecessary images. The amount of this extra data has to be kept to the minimum possible to save precious transmission times. Another challenge is that the user should be given an advanced motion scheme such as ones found in 3D games. This is not present in other protocols that try to predict next user movements, in order to enable them to simply send all possible combinations in advance.

In the next chapter we introduce a new idea which promises to solve IR and POD problems and outperform them by simply making future images or least a major part of

them available on the client before needing to use them upon movement. Thus, almost eliminating network dependency needed to deliver views of new positions. The new protocol will study user movements and will adapt to their movements to improve its performance. This is a major difference between the new idea on one hand and IR and POD on the other hand, since the later don't understand user movement and do not change to better suit them.

Chapter 5

The Proposed Virtual Environment Streaming Protocol

In this chapter, we present a new proposed streaming protocol called *Partial Panorama Prediction* (*PPP*) and discuss it in details. We will show how the protocol actually *predicts* future user movements where user movement is more advanced and complex than existing solutions. The protocol does that while saving bandwidth from transmitting unnecessary data. It only streams data which is needed to generate possible next views when the user moves. Another aspect of the protocol is that it *adapts* its self to user movement. It changes its parameters and priorities in response to frequent movement types.

This chapter is divided as follows: section 5.1 describes the objectives which the partial panorama prediction protocol tries to achieve. A richer and improved user navigation scheme allowing better interaction with the virtual environment is presented in section 5.2. Section 5.3 lays out some 3D background necessary to understand the protocol. It explains what view frustum and projection plane are. Section 5.4 introduces the idea of partial cubic panoramas which saves bandwidth and improves system response. Section 5.5 introduces a new method for transmitting imagery for a type of movement called *strafe*. It explains the idea and how it is more efficient than normally streaming new

data upon request. It proves that the new idea streams less data and streams it before it is actually needed. Hence, this idea is used by the *PPP* protocol to predict future user movements. Section 5.6 expands on the strafe transmission idea by transforming any movement in any direction to a strafe movement and therefore utilizing this powerful technique. Section 5.7 formalizes *PPP* streaming protocol on the server and the client devices. Section 5.8 shows our *PPP* implementation. It also explains the different techniques and algorithms used to implement various ideas in the protocol such as partial panoramas and depth compression. Finally, current implementation problems and limitations are identified and listed to be solved in future research work.

5.1 PPP Protocol Objectives

The goal of the *PPP* streaming protocol is to enable mobile device users to access and navigate virtual environments in an efficient way that overcomes the limitations of mobile devices hardware. Unlike other existing protocols, the new protocol should provide mobile users with a rich navigation scheme and thereby a rich user experience. All of this, while maintaining interactivity and system responsiveness.

Obviously when user navigation becomes more complex, system responsiveness will be reduced because of the emerged need of transmitting larger number of images and the fact that predicting user movements becomes much more difficult than the case of simple predefined movements. We should be able to predict user movements and deliver new imagery prior to needing them.

In the following section, we will show the improved user navigation which is allowed in the proposed protocol.

5.2 User Navigation Scheme

The proposed navigation scheme gives the user a great deal of flexibility in navigating the virtual environment. In earlier ideas that try to predict future user movements, user navigation was restricted to simple and predefined movements so that the system could predict and pre-stream all or most likely anticipated images. This is a major limitation that our approach tackles. We will provide the user with a navigation scheme which tries to emulate a player's movement in a typical first shooter game. User navigation is described as following:

- The user can move forward and backward by a fixed step size D.
- The user can move left or right (strafe) by a fixed step size D.
- User movement is restricted to the XZ plane. The user can not change altitude i.e. elevate, jump or crouch.
- The user can look up and down freely and by any angle.
- The user can rotate horizontally (around y-axis) freely and by any angle.

This movement scheme is much more complicated than simple movement schemes proposed in previous works and should allow much more interactivity between the user and the system.

In the following sections, we will illustrate the basic ideas on which the protocol is based, but before that, let's dive into some 3D mathematics necessary to understand the protocol.

5.3 3D Mathematics Background

To understand the protocol, we need to elaborate on key 3D mathematics related to cameras and user navigation.

5.3.1 View Frustum

According to [1], we define a *view frustum* as the volume of space containing everything that is visible in a 3D scene. The view frustum is shaped like a pyramid whose peak lies at the camera position as shown in Figure 5.1.

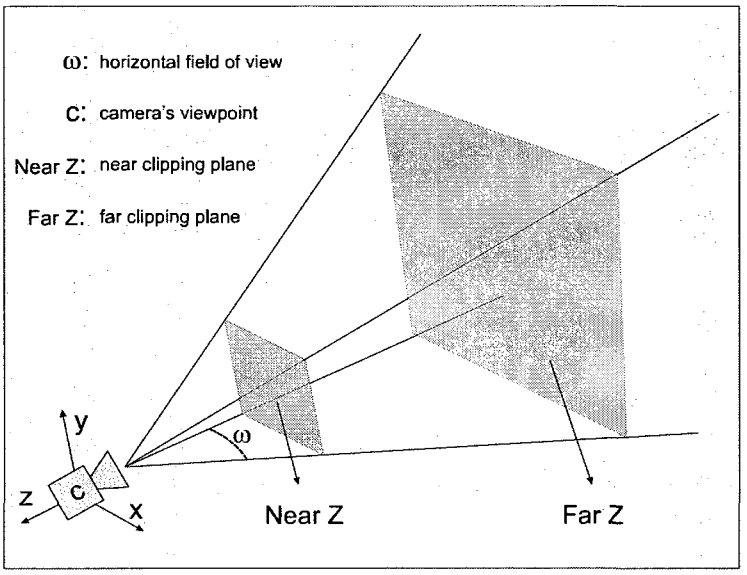


Figure 5.1: View Frustum.

A view frustum is defined by the following:

- Near Z plane: minimum distance at which objects are visible.
- Far Z plane: maximum distance at which objects are visible.
- Horizontal field of view angle ω.
- Aspect ratio: the ratio of ω to Vertical field of view angle.

5.3.2 Projection Plane

According to [1], a *projection plane* is a plane perpendicular to camera's viewing direction and lies at distance e from the camera where left and right frustum planes intersect it as $x = -1$ and $x = 1$ respectively. This is illustrated in Figure 5.2.

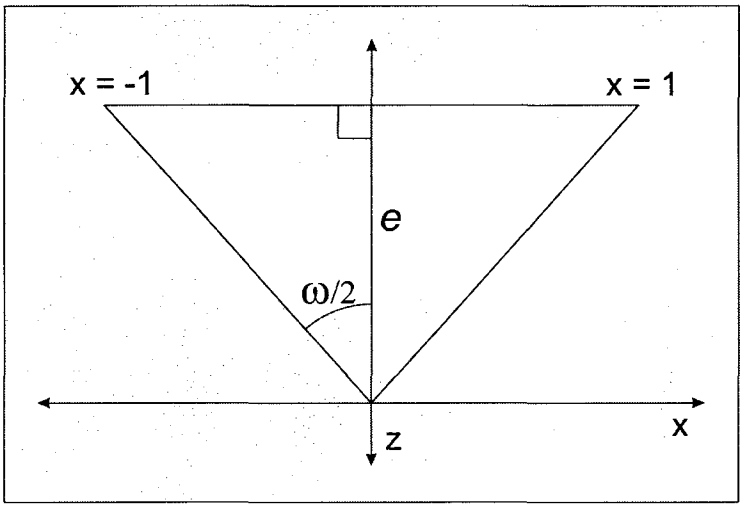


Figure 5.2: Projection plane and the focal length e.

e is called the *focal length* of the camera and is inversely proportional with ω. Using simple geometry, e can be computed as:

$$e = \frac{1}{tan(\frac{\omega}{2})} \tag{5.1}$$

Projection is rendering a 3D scene into a 2D display. We need to determine where on screen each vertex should be drawn. A 3D point $P(x, y, z)$ is projected on the projection plane as follows:

$$x = \frac{e.P_x}{P_z}, \quad y = \frac{e.P_y}{P_z} \tag{5.2}$$

The formulas shown in this section will be a base to an idea that *PPP* protocol will use to predict future user movements, but before that, let's see how the protocol predicts future rotations. It does that differently than *POD* . Instead of streaming a full

panorama, only parts of the panorama that will be most probably used are streamed. The key idea is to guarantee a minimum rotation coverage for which views are available at the client at all times. This is the first major new idea in *PPP* that strives to increase locality and therefore reduce newly requested views' response times.

5.4 Partial Panoramas

Panoramas can be used as means to predict future user rotations. By streaming a panorama for a position in the virtual environment, we are localizing all rotations at that position. However, transmitting complete panoramas for every position the user navigates to is inefficient as seen in *POD* protocol since the size of a panorama is large. Another important factor is that most of the time, the user will not use the entire 360° × 360° view offered by the panorama. This means most of the panorama has not been used and was unnecessary.

An idea that comes to mind is, what if we find a way to reduce the amount of panorama data that will not be used and at the same time stream parts which are most likely to be used? This means lower bandwidth waste and faster response times because we stream less data and more importantly we stream data that is considered most probable to be used in future rotations. Thereby making these rotations' response times very small since requested data is already localized at the client.

We introduce the idea of *partial panoramas*. We only stream parts of the view the user will most likely need. These parts are simply the closest views the user can see from his current looking direction. Instead of sending images that cover 360 degrees of horizontal and vertical rotations, we send images that cover θ horizontal rotations and β vertical rotations; where: $0 \leq \theta \leq 360$ and $0 \leq \beta \leq 180$. When θ is 360 and β is 180, this means a complete panorama exactly as used by *POD* . 0 values for θ and β means we only stream the current view and we do not consider any rotations. This is exactly what *IR* protocol does. In a sense, this idea includes both extremes; *IR* and *POD* and the

in between compromises between bandwidth saving and future rotations localizations. We always guarantee that up to θ degrees of horizontal rotations and up to β degrees of vertical rotations are present on the client. This area of rotations is considered the most probable the user will need since they are the closest to his viewing direction. The user can freely rotate within this area with out having to request the server to stream new images. Once the user has rotated close enough to exceed this area, the system sends new images necessary to maintain the coverage. Figure 5.3 shows the (θ, β) rotation coverage from where the user is facing. The X-axis represents horizontal rotations and the Y-axis represents vertical rotations. The center point of the coverage rectangle is the direction where the user is facing. Only the small rectangle is streamed to the client as opposed to the big rectangle which represents a complete panorama.

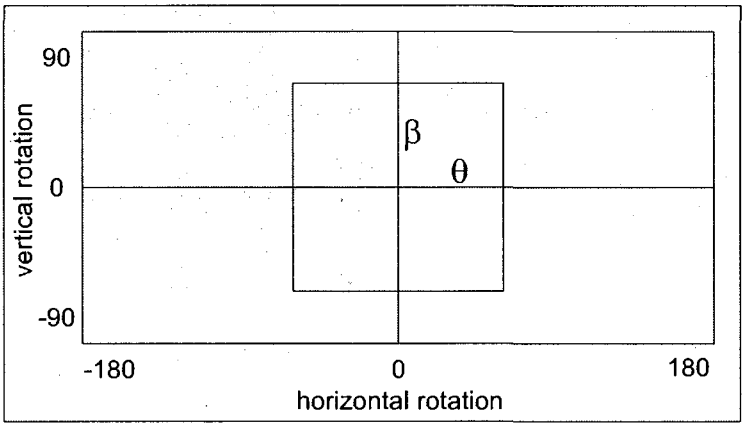


Figure 5.3: (θ, β) rotation coverage.

Figure 5.4 shows what happens when the user rotates *significantly* in a direction, the difference between the old and new coverage rectangles is streamed to maintain this coverage.

When do we consider the user too close to exceed the coverage? This depends on the application of the protocol. As can be seen, this idea saves valuable bandwidth and streams only parts which will most likely be needed in future rotations at a position in the virtual environment. The values of the (θ, β) pair play an extremely important role

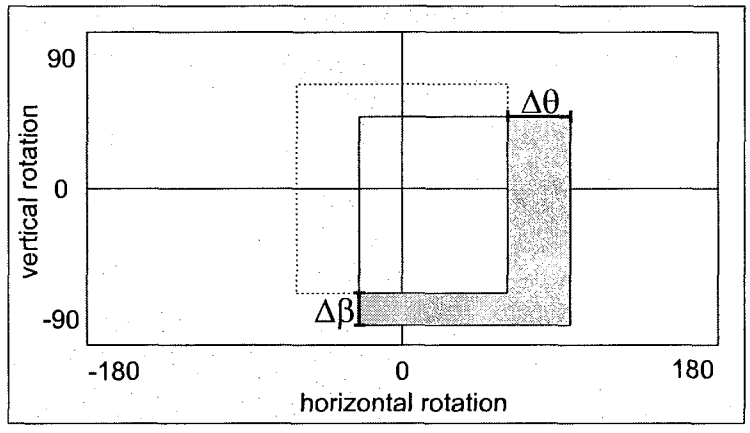


Figure 5.4: Difference between old and new rectangles is streamed upon user rotation.

in the efficiency of the prediction. Large values mean we predict the user will rotate a lot while small values indicate that we predict the user will not rotate that much and therefore streaming unnecessary parts is avoided. The tradeoff is in the frequency we need to ask the server to stream new views. In the former case, we stream more data to increase localization and decrease server dependency while in the later we try to save bandwidth but we consult the server more in case the user rotated a lot. The optimal values can be selected depending on the application or by adapting the system to user movement patterns. This idea of making the system intelligent and adaptive to user movement will be thoroughly discussed as we formally specify *PPP* protocol.

5.5 Strafe Movement Streaming

A strafe is a simple translation along the camera's X-axis. It does not involve any rotation. Figure 5.5 illustrates this movement.

Instead of transmitting a new partial panorama from the server when a strafe occurs; the idea is to stream only the *lost pixels* of the current viewable image at the client. Other parts of the panorama will be streamed normally.

When the user strafes by D units; a 3D point $P(x, y, z)$ becomes $P'(x + D, y, z)$. By

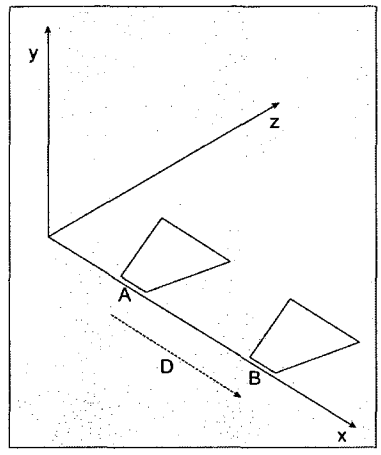


Figure 5.5: Strafe by D units from position A to position B.

applying formula 5.2 to P', its corresponding pixel (x', y') becomes:

$$\begin{aligned} x' &= \frac{e(P_x + D)}{P_z} = x + \frac{e.D}{P_z} \\ y' &= \frac{e.P_y}{P_z} = y \end{aligned} \tag{5.3}$$

$\frac{e.D}{P_z}$ is the pixel displacement. Let's denote it by D'.

To be able to compute D', we need to know the depth of each pixel. Therefore pixels' depth values must be sent to the client. This is an overhead but it will pay off as we will show in a bit.

We can roughly estimate the percentage of lost pixels caused by a D units strafe assuming that P_z is identical for all pixels using this formula:

$$L = \frac{D'}{W} \tag{5.4}$$

where L is the percentage of lost pixels and W is the image width, measured in pixels. This formula is useful for estimating the worst case loss percentage by assuming $P_z = nearZ$ for all pixels. Clearly D' is directly proportional with e and D and is

inversely proportional with P_z. The loss percentage depends on these three inputs.

Let's have a worst case scenario using typical values for our inputs:

- P_z is between $nearZ$ and $farZ$. The worst case is when all the pixels have a depth of $nearZ$ and that $nearZ = 1$;
- $\theta = 90$ degrees, therefore $e = 1$;
- a strafe of 10 units, $D = 10$;
- and image width $W = 200$ pixels.

Applying formula 5.4 yields a loss percentage of:

$$L = \frac{D'}{W} = \frac{10}{200} = 5\% \; loss$$

We have saved transmitting 95% of the *current viewable image* in worst case. Therefore waiting time is greatly reduced when the user strafes. Instead of waiting for a new image to arrive, we already have 95% of it available and we only need to stream the lost 5%. (provided that depth values at the current position are available). To compute the overall percentage of bandwidth saving, we must take into account the depth values that were transmitted with the first image as well as the depth values that will be transmitted with the new image parts.

In the normal approach we need to stream two images; one for the current view and one for the new view after the strafe. In our approach, we stream the first image along with its depth values. Assuming depth values are 25% the size of the image, this can be estimated to 1.25 images. When a strafe occurs, we stream 5% pixels of the new image and their depths. That means we stream 5% (pixels) +25% * 5% (depth of these pixels) which equals to 6.25% of the image size. Overall, we have streamed 125% + 6.25% = 131.25% instead of 200% image size. *The saving percentage* 34.4% *in the worst case.* It is important to differentiate between the benefits we gained using the strafe idea:

1. Overall bandwidth saving; which is 34.4% in our example.

2. System response time; in our example we have to wait for 5% of the new image to complete the new image while in the normal case we have to wait for an entire image to arrive. We have streamed less data and at the same time in priori. This results in a massive improvement in system responsiveness.

The overhead suffered from transmitting depth values was compensated by saving a huge amount of imagery that was being sent upon a strafe. Another factor is streaming depth in advance improves response time for generating the new image. Figure 5.6(a) shows the view at the client's current position and 5.6(b) shows the generated view from a strafe to the left. The green pixels are the pixels lost due to the strafe and need to be streamed to the client. The amount of lost pixels in this example is 4.53% i.e. 95.47% of the new image was generated on the client and need not to be streamed.

(a) (b)

Figure 5.6: Original and strafe generated images.

The idea of strafe transmission will prove to be a key corner stone in *PPP* protocol. The next step is to find a way to transform *any movement* to a strafe movement and

therefore generalize this idea over any type of movement.

5.6 Directional User Movement Streaming - Key Partial Panoramas

The next question to answer is how to generate an image for the user when he rotates, i.e. changes where he looks, and then moves in the new direction. We introduce a notion called *key partial panorama*; which is a partial panorama at a certain position that is used to generate views at other positions and rotations. Figure 5.7 visualizes the idea. The user is at the center of the circle. The circle's perimeter represents positions the user might navigate to from his current position, obviously by rotating in a direction and moving forward. The user initially faces the direction of position A. He then rotates by an angle of θ and moves forward. Normally, we would need to stream a partial panorama at position B centered towards orientation θ.

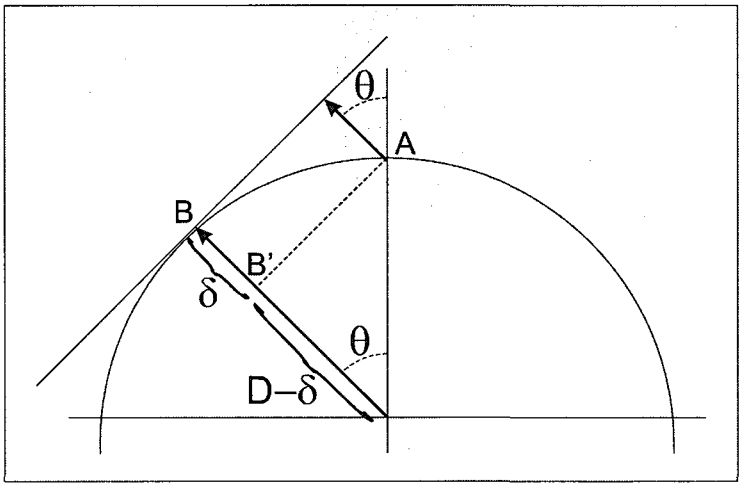


Figure 5.7: θ° rotation followed by a forward movement by D units to position B.

The quick strafe movement idea however can be applied to this situation. We can transform the original problem to a strafe problem for which we already developed an efficient solution. Suppose the key partial panorama was at Position A in the figure.

If this panorama covers the orientation θ i.e. we can rotate towards orientation θ at position A; then we can strafe from A to a position B' *close to* B. There will be a margin of error δ between the intended position B and the strafe generated position B'.

Acceptable error percentage $\frac{\delta}{D}$ depends on the application. Let's calculate δ and discover on what values it depends. Figure 5.8 shows the solution.

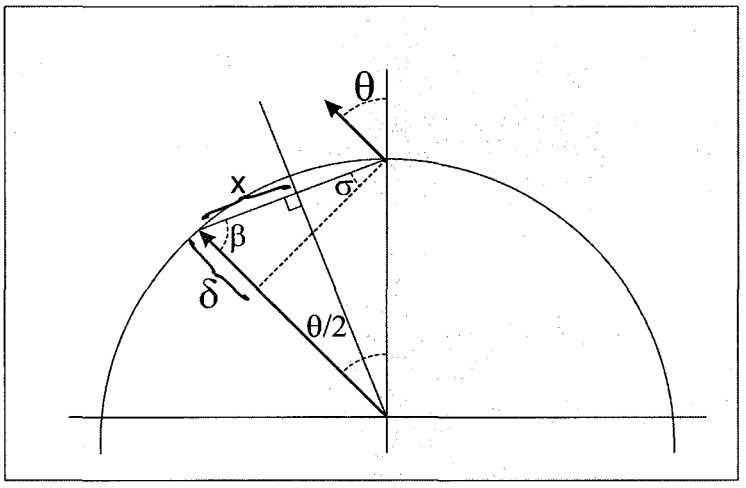


Figure 5.8: δ calculation.

Observing Figure 5.8 and using simple trigonometry, we can derive the following:

$$\begin{aligned}
(\frac{\theta}{2}) + \beta &= 90 \\
\sigma + \beta &= 90 \\
\sigma &= (\frac{\theta}{2}) \\
Sin(\frac{\theta}{2}) = \frac{x}{D} &\rightarrow x = DSin(\frac{\theta}{2}) = DSin(\sigma) \\
Sin(\sigma) = \frac{\delta}{2x} &\rightarrow \delta = 2xSin(\sigma) = 2DSin(\sigma)^2
\end{aligned} \tag{5.5}$$

According to equation 5.5, it can be seen that δ increases as $\frac{\theta}{2}$ (or σ) increases since it depends on its Sin. Table 5.1 shows the error percentage of the total step size for different θ rotations from the key panorama.

Table 5.1: Error percentage according to the step size.

θ **(degrees)**	$\frac{\delta}{D}$ **(error percentage)**
45	29.3%
40	23.4%
30	13.4%
22.5	7.6%
20	6%
15	3.4%
11.25	1.9%
10	1.5%
5	0.3%
4.5	0.3%
1	0.15%

The smaller the rotation, the smaller error margin δ, which infers more accurate results. If we have a key partial panorama at where the user moves forward in the direction of a horizontal angle λ and vertical angle μ, and this partial panorama covers θ horizontal rotations and β vertical rotations, we can generate viewable parts of panoramas for any forward movement in any direction between $\lambda - \theta$ and $\lambda + \theta$ horizontally and any direction between $\mu - \beta$ and $\mu + \beta$ vertically. The non viewable parts of the panorama at the new position are streamed normally.

The value of θ is extremely important since it determines the size of the key partial panorama and the number of key partial panoramas needed to cover movements in all directions. We limit θ to be from 5 to 45 degrees. θ values larger than 45 will result in big displacement errors and θ values less than 5 does not achieve the goal of transmitting key imagery to generate intermediate images since nearly all images will be streamed.

The server can change θ values according to client movements. This means that the server adapts to client's movement pattern. If the client rotates little, then θ should be reduced to save streaming unnecessary panoramic portions. If the client rotates a lot then θ should be increased to cover larger user rotations. The maximum value of θ depends on the application and what margin of error is deemed acceptable.

Pre-streaming a number of key partial panoramas that covers most probable next

user movement is how *PPP* protocol predicts next user movements. Depending on user movement, new key partial panoramas will be streamed in order to cover possible next movements. In case the system decided to ignore streaming some key partial panoramas for the sake of prediction; i.e. the user does not go in that direction frequently, this would be considered a miss. The new imagery will be normally streamed and the protocol will record this event and consider it in future user movements. Now we are ready to assemble all the ideas we discussed so far and formally layout the partial panorama prediction protocol.

5.7 PPP Protocol Formalization

All of the previous sections introduced ideas that will be used by the *PPP* protocol. The protocol depends on partial panoramas, strafe, key partial panoramas, user movement and user movement history.

PPP protocol is *adaptive*, meaning that it behaves according to user's movements. Certain imagery is pre-streamed since they will be most likely used in the user's next movements. Others will be streamed if the need for them arises. The protocol is always one -or more- step(s) ahead of the user since it pre-streams necessary images to cover all possible movements in advance and in an efficient and bandwidth utilizing way.

A *look-ahead* strategy will be employed. The server should decide the value of the look-ahead depending on the situation. For example a look-ahead of value 2 means that the server pre-streams data needed to cover 2 movements. This value depends on the consistency of user movement. If the user changes his route a lot then the value of the look-ahead should be decreased.

Initially, the user is located at certain position (x_0, y_0, z_0) in the virtual environment looking at (oriented towards) horizontal angle λ and vertical angle μ. λ is the rotation around the camera's y-axis and μ is the rotation around camera's x-axis.

In Figure 5.9, this location is the center of the circle and the user is facing the front

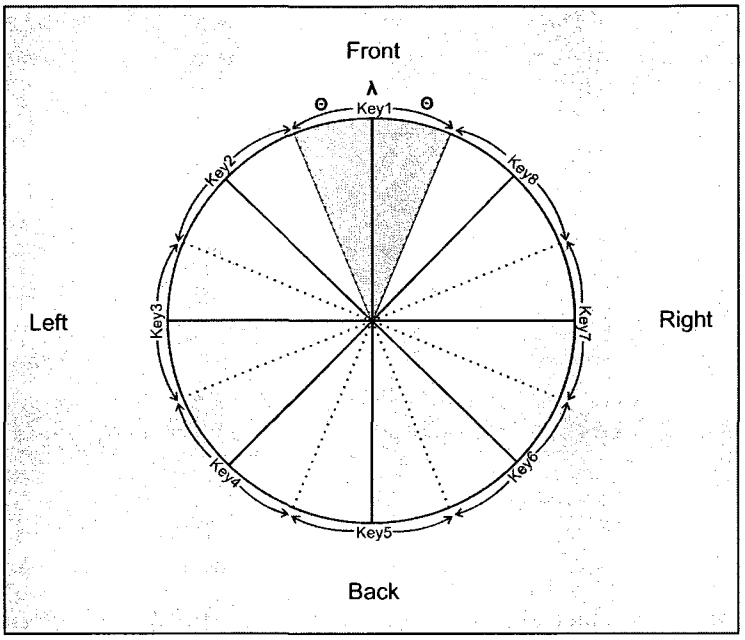


Figure 5.9: PPP protocol explanation.

direction. The user can rotate by any horizontal angle and move in that direction i.e. it can move to any point on the circle's perimeter. As seen from the figure, each key panorama covers an arc of the perimeter.

The following variables are used by *PPP* protocol algorithms:

Algorithm 1 COMMON VARIABLES

▷ **Variables:**
1: θ {the angle up till horizontal user rotation is covered by a key partial panorama. Ranges from 5 to 45 degrees.}
2: β {the angle up till vertical user rotation is covered by a key partial panorama. Ranges from 5 to 90 degrees.}
3: D {user step size.}
4: $UserOrientation$ {horizontal and vertical angles the user is facing.}
5: $CurrentPanorama$ {Partial panorama at user's current position.}
6: $FrontKeyPanorama$ {Partial panorama used to generate intermediate views if the user steps forward.}
7: $BackKeyPanorama$ {Partial panorama used to generate intermediate views if the user steps backward.}

Algorithm 2 lays out the *PPP* server protocol.

Algorithm 2 PPP SERVER PROTOCOL

1: Save user's movement or rotation into his session and execute adaptation routine. (Ignore this step at startup).

2: Stream partial panorama at current position (x_0, y_0, z_0) - center of the circle in the the figure - covering rotations from $(\lambda - \theta, \mu - \beta)$ to $(\lambda + \theta, \mu + \beta)$.

3: Stream partial panorama at the position in front of the user, i.e. at position $(x_0, y_0, z_0 + D)$ with the same panoramic range since the user is facing the same direction. This is Key panorama 1 in Figure 5.9. This partial panorama is used if the user steps forward. It is also a key partial panorama which covers forward user movements in directions $\lambda - \theta$ to $\lambda + \theta$. set this partial panorama as $FrontKeyPanorama$.

4: Stream partial panorama behind the user, i.e. at position $(x_0, y_0, z_0 - D)$ with the same panoramic range since the user is facing the same direction. This is Key panorama 5 in Figure 5.9. This partial panorama is used if the user steps backward. It is also a key partial panorama which covers backward user movements in directions $\lambda - \theta$ to $\lambda + \theta$. set this partial panorama as $BackKeyPanorama$.

5: Wait for a client's message.

Action: {$ClientRotation$ - the client rotated}

{Check if the user is close to exceed the coverage of $FrontKeyPanorama$. The sensitivity of the check can be adjusted by the $weight$ used.}
6: **if** $(Abs(UserOrientation.H - FrontKeyPanorama.H) >= \theta \times weight)$ **then**
7: Stream new panoramic portions at the user's current position to maintain a minimum of θ degrees horizontal coverage and β degrees vertical coverage.

8: Update $UserOrientation$ to new direction.

9: Stream a new key partial panorama to cover forward movements in directions beyond $(\lambda - \theta, \lambda + \theta)$. Closest key panorama is located by moving forward in the following direction : $FrontKeyPanorama.H$ $(+or-)$ (2θ) {In Figure 5.9, this new key can be key panorama 2 or 8 depending or user's rotation direction.}

10: Stream a new key partial panorama to cover backward movements in directions beyond $(\lambda - \theta, \lambda + \theta)$. Closest key panorama is located by moving backward in the same direction as in the previous step. {In the Figure 5.9, this new key can be key panorama 4 or 6 depending or user's rotation direction.}
11: **end if**

{Check if the user actually exceeded the coverage of $FrontKeyPanorama$}
12: **if** $(Abs(UserOrientation.H - FrontKeyPanorama.H) >= \theta)$ **then**
13: update $FrontKeyPanorama$ and $BackKeyPanorama$ to newly sent key partial panoramas.
14: **end if**

Action: {$ClientStrafe$ - the client strafed}

15: Update $CurrentPanorama$ to new position.
16: Stream lost pixels of the viewable panorama segment and their depths.
17: Stream the rest of $CurrentPanorama$ partial segments normally to maintain (θ, β) rotation coverage.
18: Restart Server Side Protocol.

Action: {$Forward$ - the client stepped forward}

19: Update $CurrentPanorama$ to new position.
20: If a strafe occurred on the client then stream lost pixels of the viewable panorama segment and their depths.
21: Restart Server Side Protocol. In this case, we omit sending the back key panorama since $CurrentPanorama$ becomes $BackKeyPanorama$.

Action: {$Backward$ - the client stepped backward}

22: Update $CurrentPanorama$ to new position.
23: If a strafe occurred on the client then stream lost pixels of the viewable panorama segment and their depths.
24: Restart Server Side Protocol. In this case, we omit sending the front key panorama since $CurrentPanorama$ becomes $FrontKeyPanorama$.

Note: all partial panoramas are streamed with their pixel depths to enable strafing from the client.

The following adaptation routine is executed on the server to adapt to client movements.

Algorithm 3 ADAPTATION ROUTINE

1: If in recent movements, the user rotated a little and maintained a constant route, then increase look-ahead and decrease θ.
2: If the user does not navigate in a certain direction. E.g. does not move backward, then streaming key panoramas that covers backward movement can be ignored.
3: If user changes directions constantly and in big amounts then increase θ and decrease look-ahead.

Listed below is the *PPP* client protocol.

Algorithm 4 CLIENT PROTOCOL

1: **while** client is alive **do**
2: Wait for user's next movement.

Action: {*Rotation* - the client rotated}
3: inform the server.

Action: {*Strafe* - the client strafed}
4: apply strafe movement algorithm to *CurrentPanorama* to generate the new viewable panorama segment at $(x_0 + D, y_0, z_0)$ or $(x_0 - D, y_0, z_0)$ depending on the direction of the strafe (left or right).
5: display the new generated segment to the user.
6: Send the indices of lost pixels to the server and inform it of a strafe movement.

Action: {*Forward/Backward* - the client stepped forward of backward}
7: **if** there is a key partial panorama that covers movement in the current direction **then**
8: **if** user is facing a key partial panorama **then**
9: set this key partial panorama as *CurrentPanorama* and display it to the user.
10: **else**
11: strafe from *FrontKeyPanorama* or *backKeyPanorama* depending on the movement and generate the new viewable panorama segment and display it to the user.
12: Send the indices of lost pixels to the server.
13: **end if**
14: **else**
{this case is resulting from protocol prediction that ignored sending the key partial panorama, this is considered a miss. Nothing can currently be displayed to the user.}
15: **end if**
16: Inform the server of the movement.
17: **end while**

It is evident that prior to user movements or rotations, the requested imagery is available on the client cache in full or at least most of it can be locally generated using the strafe algorithm. *PPP* protocol thus achieves the prediction of an advanced user scheme by using a partial panorama that covers most probable rotations at the current

position and by using a set of key partial panoramas which cover movements in directions closest to the user's current orientation. The strafe algorithm has solved the problem of predicting movements in any given direction efficiently and by only sending a small set of key imagery. The prediction now is adapted by changing the partial panoramas coverage and by sending key partial panoramas in directions most probably the user will navigate to. This is much simpler, more feasible and efficient than sending images for every possible future movement which worked with simple movement patterns.

In the next section, we describe our implementation of the *PPP* protocol. Like any implementation, this implementation is not perfect. We propose solutions and algorithms for *PPP* protocol techniques such as depth compression, partial panoramas implementation, the type of panoramas to use and strafe algorithm. These issues and many more are implementation specific. *PPP* protocol performance is greatly impacted by changing these implementation decisions. We describe our current solutions and algorithms and layout current implementation problems, limitation and suggested enhancements to improve *PPP* protocol implementation in the future. As will be seen in the performance evaluation later in this thesis, our *PPP* implementation was able to outperform *IR* and *POD* in terms of many metrics, most important of which is the *waiting time* for the new view to arrive, thus achieving better user experience alongside a better navigation pattern.

5.8 PPP Protocol Implementation

We use the cubic panorama format. This type of panoramas is the simplest. It is easier to render and more efficient than using spherical or cylindrical panoramas and does not produce image distortions such as suffered by the last two. Rendering cubic panoramas was explained in chapter 4. A cubic panorama consists of six Images wrapped on a cube. Each image is taken by a $90^\circ \times 90^\circ$ field of view camera and is orthogonal to other faces, therefore covering all the views from a certain position in the virtual world. Figure 5.10

shows a cubic panorama unfolded.

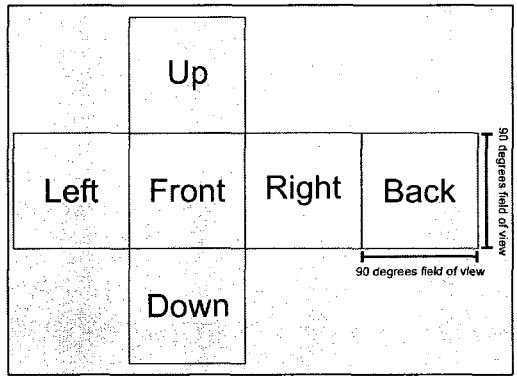


Figure 5.10: Unfolded cubic panorama.

We define a *cubic panorama axis system* to be all the directions the user can face from a point in the virtual environment. As shown in Figure 5.11, the X-axis represents horizontal orientations the user may take and the y-axis represents vertical orientations. Initially, the user faces the center of the front face i.e. rotation pair $(0, 0)$. As the user rotates and depending of the field of view, other cube faces cover new views. The cubic panorama covers a 360 degree horizontal and vertical rotations as shown in the same figure.

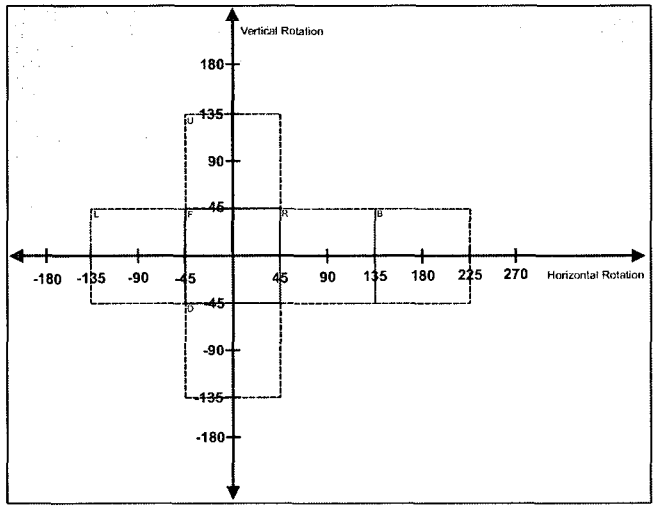


Figure 5.11: Cubic panorama axis system.

The implementation of the cubic panorama is shown in Figure 5.12. *CubicPanoramaFace* enumerates all cube faces. The *CubicPanorama* class contains a hash table of cube faces. Each face has an image to store the pixel data and a depth matrix to store pixel depths needed for strafing.

We have now shown that we use cubic panoramas and have seen how we implement them. The next step is to show the implementation of partial panoramas.

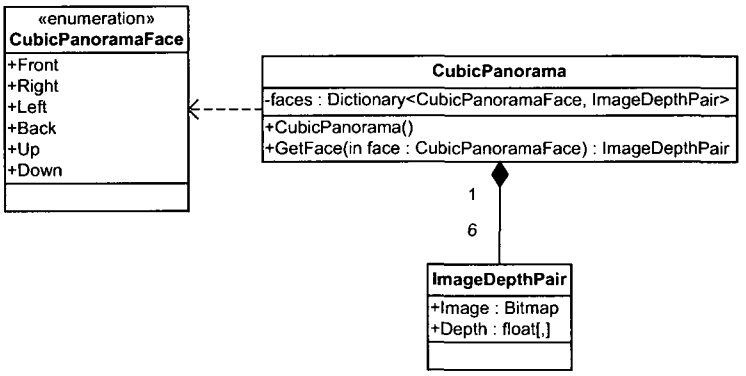


Figure 5.12: Cubic panorama implementation.

5.8.1 Partial Panoramas Algorithm

Partial panoramas cover a subset of the entire view space. The implementation of partial panoramas has to identify and stream parts of the cubic panorama shown in Figure 5.11 that cover user's current view and up to θ of horizontal rotations and β of vertical rotations from his current view. Another issue is how to send only the difference between the current view and the new orientation view. This is crucial to avoid sending the same data more than once. A user's current view and possible (θ, β) rotations coverage can be mapped to the cubic panorama axis system developed as depicted in Figure 5.13.

The small solid rectangle is the user's current view. The user faces the $(0, 0)$ direction and has a field of view of 45 degrees both horizontally and vertically. the dotted rectangle contains parts needed to achieve a coverage of 45 degrees in both horizontally and vertically. We have to stream the whole front face and the parts of right, left, up

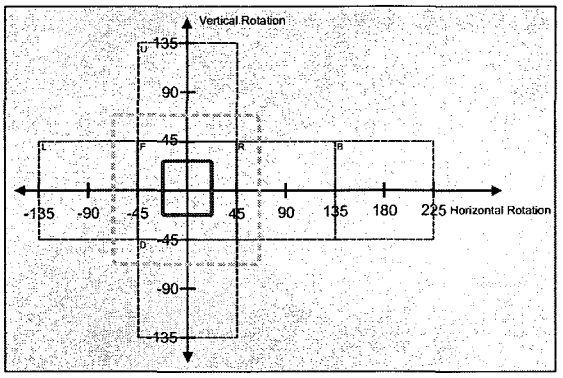


Figure 5.13: Partial Panorama for current user view.

and down faces contained in the dotted rectangle. There are four areas outside the cubic panorama representation we currently have; which is explained by the fact that the up and down faces are shared by all front, left, right and back faces. For instance, When the user is facing the right face and rotates up, he sees the up face but from the right edge instead of the bottom edge if he faced the front face. What happens is that the up and down face images are simply rotated when the user horizontally rotates. In our previous example, the up face was rotated by 90 degrees clockwise when the user faced the right face. Figure 5.14 extends our panorama representation by adding extra faces derived by rotating the *up* and *down* faces depending on the horizontal direction of the user.

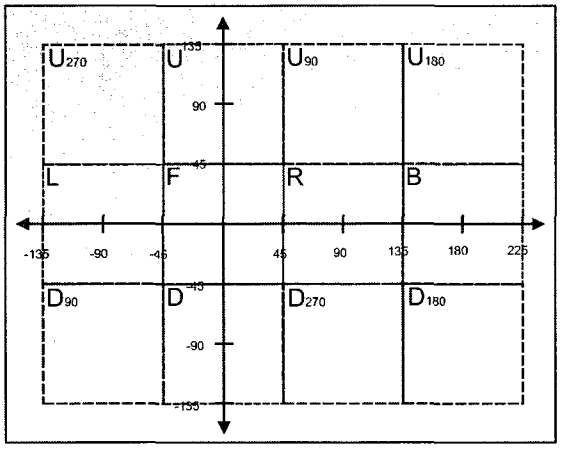


Figure 5.14: Extended cubic panorama representation.

The new faces (dotted for clarity) are derived from up and down faces. Up90 is derived by rotating the up face by 90 degrees clockwise, up180 is derived by rotating the up face by 180 degrees clockwise and so on. Note that the down face rotates in the opposite direction of the up face. By mapping the same partial panorama in Figure 5.13 to the extended representation we get the following as in Figure 5.15(a).

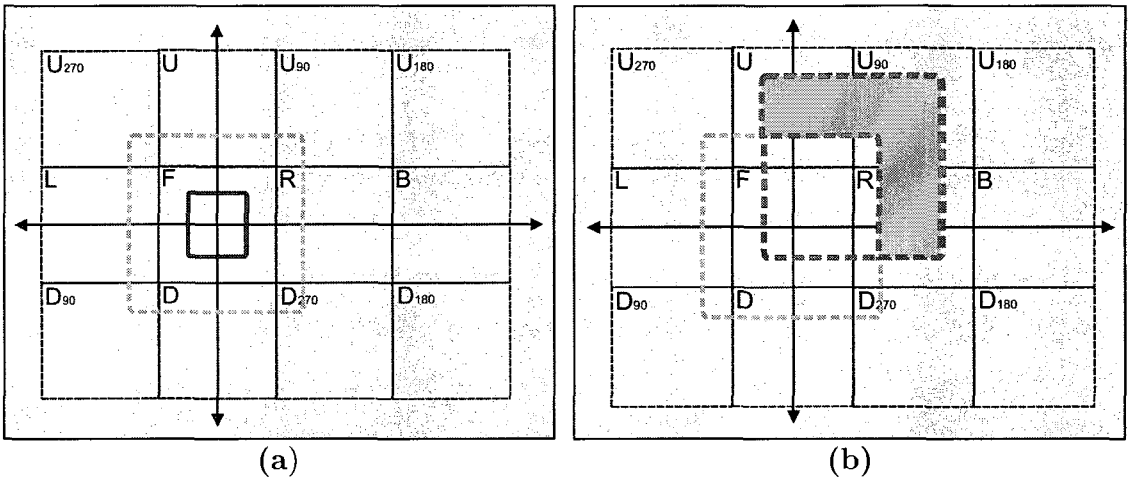


Figure 5.15: Partial panorama on the extended panorama representation.

All the required views are now contained in the new panorama representation. For every face, we have to determine the parts to send and keep track of sent parts to avoid sending them again. For example, if the user is now facing direction $(45, 45)$, we have to send the difference between the new coverage and the already sent parts as Figure 5.15(b) depicts. Rotated up and down faces have to be handled in a different way. After determining the areas on these derived faces, this area has to be rotated in the opposite direction of the face's rotation to obtain the actual area on the original face. For instance, the area on Up90 has to be rotated by -90 degrees to transform it back to the original up face.

The theories have been identified. It is time to discuss the implementation. Figure 5.16 shows the classes that collaborate to provide partial panorama functionality.

SpatialCubeFace enumerates the six cube faces and the faces derived by rotating

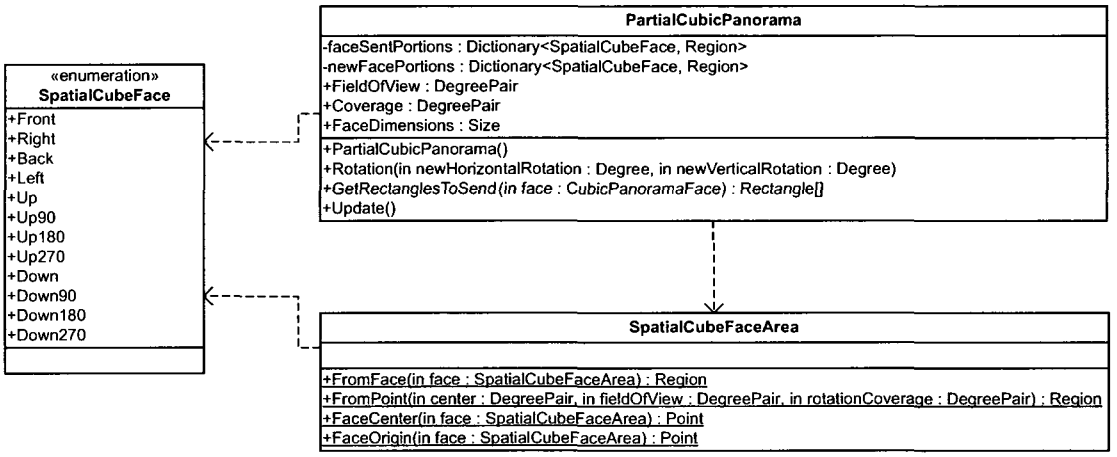


Figure 5.16: Partial panorama implementation classes.

up and down faces. The *SpatialCubeFaceArea* class provides methods that compute rectangles and points relative to the panorama axis system we developed. *FaceCenter()* method returns the center of a spatial cube face. For example the center of the front face is $(0, 0)$ and the center of the U270 face is $(-90, 90)$ -Please refer to Figure 5.14- . *FaceOrigin()* returns the upper left corner of a cube face. For example, $(45, 45)$ is the origin of the right face. *FromFace()* method returns the face's rectangle in the axis system which is simply made of the face's origin and it's size which is always 90 in width and height. *FromPoint()* method computes the rectangle centered at a certain point (rotation pair) given a field of view and the desired coverage. This method will be used to compute the coverage needed for a certain user direction.

SpatialCubeFaceArea services are used by the *PartialCubicPanorma* class which holds all the information of the partial panorama. *PartialCubicPanorma* needs to know the image size for a cube face so that it can transform degree coordinates into image pixels, the *FaceDimensions* property can be set to the face image size. Also needed is the field of view and the desired coverage. Those can be set by the *FieldOfView* and *Coverage* properties. To compute face areas needed to be streamed upon a client rotation, *Rotation()* method is called and passed the client's new rotation. *PartialCubicPanorma* automatically computes the new areas on each face that needs to be sent considering already sent

areas to eliminate streaming data more than once. After that, *GetRectanglesToSend()* method can be called to return a set of rectangles in pixel coordinates that needs to be streamed for each cube face. After sending these new parts to the client, the *Update()* method is called to add the newly computed areas to sent areas to avoid sending them in the future.

The following pseudo code shows the partial panorama algorithm implemented by the aforementioned set of classes. Please refer to Figure 5.14 for clarity.

Listing 5.1: Partial panorama algorithm implementation

```
// Variables

// regions of each spatial cube face already sent to client
Dictionary<SpatialCubeFace, Region> faceSentPortions;
// regions of each spatial cube face produced from a rotation and have not
// been sent yet to the client
Dictionary<SpatialCubeFace, Region> newFacePortions;

// Properties

// client's field of view
DegreePair FieldOfView;
// desired rotation coverage
DegreePair Coverage;
// cube face pixel dimensions, used to transform panorama
// system points to image pixels
Size FaceDimensions;

// Initilization
public PartialCubicPanorama()
{
    // initialize sent and new portions for each spatial face to empty region
    for each spatial cube face
    {
        faceSentPortions[face] = empty region;
        newFacePortions[face] = empty region;
    }
}

// Calculates new portions to be sent upon a client rotation
```

```
// newHorizontalRotation: new horizontal direction the user is facing
// newVerticalRotation: new vertical direction the user is facing
public void Rotation(Degree newHorizontalRotation, Degree newVerticalRotation)
{
    // get user's new view which is centered at the new user rotation
    // and covers the preset rotations
    Region userNewView = SpatialCubeFaceArea.FromPoint(newHorizontalRotation,
                                newVerticalRotation, FieldOfView, Coverage);

    // for each spatial panorama face, calculate the new portions
    // to send to the client
    for each spatial cube face
    {
        // get the area of the current spatial face
        Region faceArea = SpatialCubeFaceArea.FromFace(face);
        // intersect user's new view with the current face
        Region newFaceView = Intersect(userNewView, faceArea);
        // exclude already sent areas from this intersection
        newFaceView.Exclude(faceSentPortions[face]);
        // store the result in the to be sent hash
        newFacePortions[face] = newFaceView;
    }
}

/// returns a set of rectangles (in pixels) to be sent for a given cube face
private RectangleF[] GetRectanglesToSend(CubicPanoramaFace face)
{
    Region totalFaceArea = null;
    Matrix rotationMatrix = new Matrix();
    Matrix translationMatrix = new Matrix();
    SpatialCubeFace spatialFace = (SpatialCubeFace) face;

    // if the requested face is not UP or Down then return its area
    if(face != CubicPanoramaFace.Up && face != CubicPanoramaFace.Down)
    {
        totalFaceArea = newFacePortions[face];
    }
    // other than that, we have to union the face and its rotation faces areas
    // (e.g. Up and its Up90, Up180, Up270 faces)
    else if(face == CubicPanoramaFace.Up)
    {
        // add the Up new areas to total area to send
```

```
        totalFaceArea = newFacePortions[SpatialCubeFace.Up];

        // Transform Up90 new areas to match the Up face
        Region upBy90 = newFacePortions[SpatialCubeFace.Up90];
        // rotate by -90 around Up90 face center
        rotationMatrix.RotateAt(-90, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Up90));
        upBy90.Transform(rotationMatrix);
        // translate by -90
        translationMatrix.Translate(-90, 0);
        upBy90.Transform(translationMatrix);

        // Transform Up180 new areas to match the Up face
        Region upBy180 = newFacePortions[SpatialCubeFace.Up180];
        // rotate by -180 around Up180 face center
        rotationMatrix.RotateAt(-180, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Up180));
        upBy180.Transform(rotationMatrix);
        // translate by -180
        translationMatrix.Translate(-180, 0);
        upBy180.Transform(translationMatrix);

        // Transform Up270 new areas to match the Up face
        Region upBy270 = newFacePortions[SpatialCubeFace.Up270];
        // rotate by -270 around Up270 face center
        rotationMatrix.RotateAt(-270, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Up270));
        upBy270.Transform(rotationMatrix);
        // translate by 90
        translationMatrix.Translate(90, 0);
        upBy270.Transform(translationMatrix);

        // add the transformed Up90, Up180 and Up270 new areas to the total new face area
        totalFaceArea.Union(upBy90);
        totalFaceArea.Union(upBy180);
        totalFaceArea.Union(upBy270);

    }
    else if(face == CubicPanoramaFace.Down)
    {
        // add the Down new areas to total area to send
        totalFaceArea = newFacePortions[SpatialCubeFace.Down];

        // Transform Down90 new areas to match the Down face
        Region downBy90 = newFacePortions[SpatialCubeFace.Down90];
```

```
        // rotate by -90 around Down90 face center
        rotationMatrix.RotateAt(-90, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Down90));
        downBy90.Transform(rotationMatrix);
        // translate by 90
        translationMatrix.Translate(90, 0);
        downBy90.Transform(translationMatrix);

        // Transform Down180 new areas to match the Down face
        Region downBy180 = newFacePortions[SpatialCubeFace.Down180];
        // rotate by -180 around Down180 face center
        rotationMatrix.RotateAt(-180, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Down180));
        downBy180.Transform(rotationMatrix);
        // translate by -180
        translationMatrix.Translate(-180, 0);
        downBy180.Transform(translationMatrix);

        // Transform Down270 new areas to match the Down face
        Region downBy270 = newFacePortions[SpatialCubeFace.Down270];
        // rotate by -270 around Down270 face center
        rotationMatrix.RotateAt(-270, SpatialCubeFaceArea.FaceCenter(SpatialCubeFace.Down270));
        downBy270.Transform(rotationMatrix);
        // translate by -90
        translationMatrix.Translate(-90, 0);
        downBy270.Transform(translationMatrix);

         // add the transformed Down90, Down180 and Down270 new areas to the total new face area
        totalFaceArea.Union(downBy90);
        totalFaceArea.Union(downBy180);
        totalFaceArea.Union(downBy270);
    }

    // convert totalFaceArea to a set of rectangles
    RectangleF[] faceRects = totalFaceArea.GetRegionScans();

    // transform these rectangles from angle coordinates to pixel coordinates
    for (int i = 0; i < faceRects.Length; ++i)
    {
        RectangleF rect = faceRects[i];
        Point originPx = Point.Empty;

        originPx.X = (int)((rect.Left - SpatialCubeFaceArea.FaceOrigin(spatialFace).X) *
                           (FaceDimensions.Width / 90.0));
```

```
            originPx.Y = (int)((rect.Top - SpatialCubeFaceArea.FaceOrigin(spatialFace).Y) *
                               (FaceDimensions.Height / 90.0));

            Size sizePx = Size.Empty;
            sizePx.Width = (int)(rect.Width * (FaceDimensions.Width / 90.0));
            sizePx.Height = (int)(rect.Height * (FaceDimensions.Height / 90.0));

            faceRects[i].Location = originPx;
            faceRects[i].Size = sizePx;
        }

        // return cube face transformed areas
        return faceRects;
    }

    /// adds new areas to sent areas
    public void Update()
    {
        // add the new areas to sent area for all spatial cube faces and empty the new areas
        for each  spatial cube face
        {
            faceSentPortions[face].Union(newFacePortions[face]);
            newFacePortions[face].MakeEmpty();
        }
    }
```

This algorithm only approximates needed images to cover certain rotations. There will be a difference between the sent areas using this algorithm and the actual needed ones, since our algorithm is based on unfolded cubic panorama. In the future, more sophisticated algorithms that exactly compute the areas needed to achieve a certain coverage can be developed. These algorithms have to take into account the 3D mathematics involved in the cube shape of the panorama.

5.8.2 Depth Compression / Decompression

PPP protocol uses the strafe idea on the client to save streaming most of the requested images. Strafing though, requires pixel's depths in order to compute their displacements correctly. Therefore, depths are always sent with the pixels. Compressing depths can

greatly enhance *PPP* 's performance. An idea used in the implementation is *indexing.* Instead of sending a depth value for each pixel, we can take advantage of pixels having the same depth. If we some how stored the depth in an index list and encoded its position in the list into the pixels data, we can save a lot of bandwidth and speed up the *PPP* protocol.

Pixel data is encoded as PNG. PNG is a lossless compression algorithm, it preserves the exact pixel data after decompression. If we find a way to encode the depth's position into the pixel before compressing the image as PNG, then we can recover the depth and use it on the client. Prior to compression, we use a 32 bit per pixel (bpp) bitmap format. Each pixel is represented by 4 bytes holding its red, green as blue color components as well as an additional byte for the pixel's transparency (alpha channel). We can use the transparency byte and store the depth's index in the depth list. However, since we are only using 8 bits, we can only encode $2^8 = 256$ positions in the pixel. Most likely, we will need a list more longer than that to store image depth indices. We can use to two least significant bits of the red, green and blue channels to extend the size of the list. This means, the pixel will be modified, but since we are only using the least two significant bits, the difference will most likely be unnoticeable. The worst case is a channel difference of 3 if the two bit were zeros and were set to ones after the encoding of the depth or vise versa. Figure 5.17 shows the channels of a pixel and the bits we use to encode depth index.

Using this technique, we can index up to 16384 depths. What if there were more distinct depth values than this number? The answer is that we reserve the largest number we can encode; 16383, to indicate that the depth of the pixel was not encoded. Upon decompression, an overflow index is positioned at 16383 and is incremented each time it encounters a pixel with this special depth. This means the the behavior is now as the normal case with no compression at all. Our implementation uses the six bits of the color channels to store the depths segment index and the eight alpha channel bits stores the depths offset. The depth list is divided into 64 segments each has a length of 256.

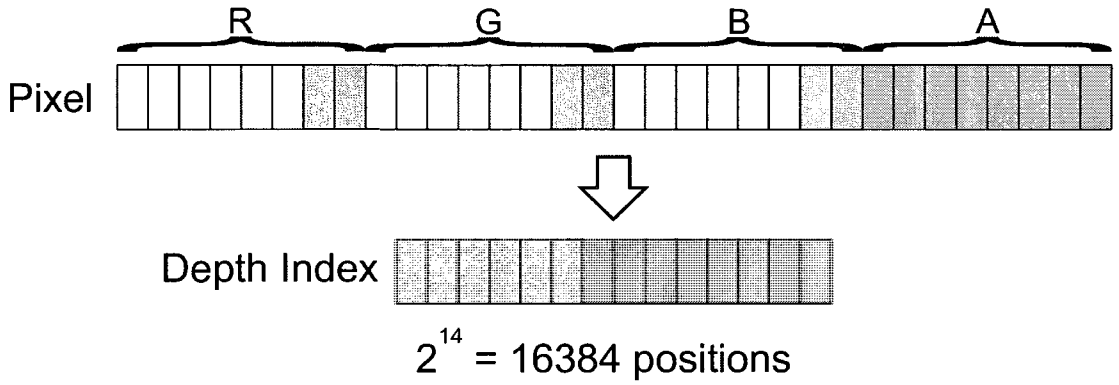


Figure 5.17: Depth index compression into pixel data.

A depth's index can be obtained by multiplying it's segment number by 256 and adding the offset. Listed below are the compression and decompression algorithms in details.

Listing 5.2: Depth Compression Algorithm

```
// encodes depth values into an image and returns a depth index list
// color: image to be encoded into
// depth: depth values for each pixel
// area:  area of the image to encode
// returns: depth index list
public static List<float> Encode(ref Bitmap color, float[,] depth, Rectangle area)
{
    // depth list
    List<float> depthIndex = new List<float>();
    // hash table that maps a depth value to a position in the depth list
    Dictionary<float, int> depthToIndex = new Dictionary<float, int>();

    for (int i = area.Top; i < area.Top + area.Height; ++i)
    {
        for (int j = area.Left; j < area.Left + area.Width; ++j)
        {
            if (!depthToIndex.ContainsKey(depth[i, j]))
            {
                // if the depth of the current pixel is new, then add it to the depth index list
                depthIndex.Add(depth[i, j]);
                // add it to the hash as well
                depthToIndex.Add(depth[i, j], depthIndex.Count - 1);
            }
            else if (depthToIndex[depth[i, j]] >= 16383)
```

```
{
    // if the depth's position in the list is beyond the maximum limit then add
    // it normaly to the list
    depthIndex.Add(depth[i, j]);
}

// depth index list will be divided into 64 segments. each segment
// is 256 entries long. Pixel's depth can be located by it's segment
// and offset. the segment will be encoded into the 2 least significant
// bits of R, G, B components of the pixel. the offest will be stored
// in the A component.
byte segment;
byte offset;

if (depthToIndex[depth[i, j]] >= 16383)
{
    // depth index cannot be encoded, set the segment to 63 and the position to 255
    // this indicates to the decoder to advance its overflow index and read
    // the next depth value
    segment = 63;
    offset = 255;
}
else
{
    segment = (byte) (depthToIndex[depth[i, j]] / 256);
    offset = (byte) (depthToIndex[depth[i, j]] % 256);
}

Color currentPixel = color.GetPixel(j, i);

// clear the 2 least significant bits of R, G, B components
currentPixel.R &= 0xFC;
currentPixel.G &= 0xFC;
currentPixel.B &= 0xFC;

// blue will store the 2 right most bits of the segment
currentPixel.B |= (byte)(segment & 0x03);

// green will store the next 2 bits of the segment
currentPixel.G |= (byte)((segment & 0x0C) >> 2);

// red will store the next 2 bits of the segment
```

```
            currentPixel.R |= (byte)((segment & 0x30) >> 4);

            // A will store the depth's offset from the beginning of its segment
            currentPixel.A = (byte)(offset);

            color.SetPixel(j, i, currentPixel);
        }
    }

    return depthIndex;
}
```

Listing 5.3: Depth Decompression Algorithm

```
// decodes depth values from an encoded image
// color: image from which depth values will be decoded
// depthIndex: depth index list
// depthValues: matrix to write depth values to</param>
// area: area of depthValues to write decoded depths to
public static void Decode(Bitmap color, List<float> depthIndex, ref float[,] depthValues,
                          Rectangle area)
{
    int overflow = 16383;
    int depthRow = (area == null) ? 0 : area.Top;
    int depthColumn = (area == null) ? 0 : area.Left;

    for (int i = 0; i < area.Height; ++i)
    {
        for (int j = 0; j < area.Width; ++j)
        {
            Color pixel = color.GetPixel(j, i);

            // read the depth segment from the RGB components
            byte segment = Convert.ToByte(((pixel.R & 0x03) << 4) | ((pixel.G & 0x03) << 2)
                                                                   | ((pixel.B & 0x03)));
            segment &= 0x3f;

            byte offset = pixel.A;
            int listPosition = segment * 256 + offset;
            listPosition = (listPosition == 16383) ? overflow++ : listPosition;
            depthValues[depthRow, depthColumn] = depthIndex[listPosition];
            depthColumn++;
        }
```

```
            depthRow++;
            depthColumn = area.Left;
        }
    }
```

These algorithms are implemented by the DepthMapCoder class shown in Figure 5.18.

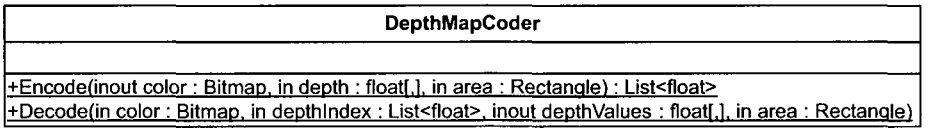


Figure 5.18: Depth compression/decompression implementation class.

To compress depths for an image, *Encode()* method is called and passed the image pixels, pixels depths, and an area in the image to compress. The method encodes the depth indices into the image and returns a depth index list. To decompress depths from an image, *Decode()* method is called and passed the image, the depth index list, and the area to decode. Also passed is a matrix of depths to be filled by the method with decoded depths.

5.8.3 Panorama Data Streaming Mechanism

We have now specified how we implemented partial panoramas and how we compress pixel depths to speed up the protocol. Our *PPP* implementation follows the formalization shown in section 5.7. The specific scenarios and use cases are already well defined there and do not have to be replicated here. An interesting piece of the puzzle still missing is how we incorporate partial panoramas and depth compression in our implementation. How do we send partial panoramas? and how does the client know how to use the incoming data and render it correctly?

Well, when the server renders a panorama and computes it's parts to send to client, and after depths are compressed, it sends a *PartialPanoramaData* message to the client

containing the following data:

- *Panorama Id:* The Id of the panorama this data segment belongs to.
- *Cube Face:* Specifies which face of the cube this data belong to.
- *Area:* Specifies exactly where on the cube face this data segment should be placed.
- *Pixels:* Depth encoded pixel data of the segment.
- *Depth Index List:* A list of depth values to be used when decompressing depth values from pixels.

Using this information, the client is able to locate the panorama, the panorama face and position on the face to update and update it with the received pixels and depths decoded using the attached depth index list.

To complete the picture, here is an example of sending a partial panorama when the user faces $(45, 10)$ direction.

Listing 5.4: Sending a partial panorama

```
// render a cubic panorama
CubicPanorama panoramaData = VirtualEnvironment.RenderCubicPanorama();

// send the ID of this panorama to the client
ServerMessage msg = new ServerMessage(ServerMessageCode.CubicPanorama, new object[] {someId});
CommAgent.SendMessage(ipEndPoint, msg);

// create and configure a partial panorama
PartialCubicPanorama partialPanoramaInfo = new PartialCubicPanorama();
partialPanoramaInfo.FaceDimensions = new Size(200,200);
partialPanoramaInfo.FieldOfView = new DegreePair(90,75);
partialPanoramaInfo.Coverage = new DegreePair(20,10);
// compute partial panorama segments for a rotation to (45,10) degree pair
partialPanoramaInfo.Rotation(45, 10);

// send partial panorama segments to client
for each (face in CubicPanoramaFace)
{
```

```
    // get current face areas to send
    RectangleF[] faceAreas = partialPanoramaInfo.GetRectanglesToSend(face);

    for each(RectangleF currRect in faceAreas)
    {
        Bitmap cubeFaceImage = panoramaData[face].Image;
        float[,] cubeFaceDepth = panoramaData[face].Depth;

        // encode this pixel's depths
        List<float> depthList = DepthMapCoder.Encode(ref cubeFaceImage, cubeFaceDepth, currRect);

        // copy current partial data
        Bitmap partialImage = cubeFaceImage.Clone(rect, PixelFormat.Format32bppArgb);
        MemoryStream stream = new MemoryStream();
        partialImage.Save(stream, ImageFormat.Png);

        ServerMessage partialPanoramaData = new ServerMessage(
                                                ServerMessageCode.PartialPanoramaData,
                                                new object[5]);
        partialPanoramaData.Body[0] = someId;
        partialPanoramaData.Body[1] = face;
        partialPanoramaData.Body[2] = rect;
        partialPanoramaData.Body[3] = (object) stream.ToArray();
        partialPanoramaData.Body[4] = depthList;

        CommAgent.SendMessage(ipEndPoint, partialPanoramaData);
    }
}

// mark the parts we just sent as sent
panoInfo.PartialPanoramaInfo.Update();
```

5.8.4 Current Implementation Status, Problems and Limitations

The current implementation is a first version prototype. Many aspects have not been optimized. Rendering cubic panoramas is done in a simple way and is currently slow. With careful techniques related to the 3D library used, rendering times can be massively

improved. More advanced algorithms for computing partial panoramas can be developed in the future. Another formats of panoramas can be used if proved to be more efficient and easier to render than cubic panoramas. The current implementation does not include adaptiveness nor look ahead strategies. All the parameters are static at the moment. This feature is left for the future for more advanced research. Problems related to image-based rendering such as *occlusion* caused by revealing objects which were hidden before moving to the new position, are not solved and are considered outside the scope of this thesis.

The implementation is still in its first version and needs more time, resources and further research to reach to a stage where it can be turned into a product and where we can realize the full potential of the *PPP* protocol ideas we have identified in this chapter. The performance evaluation done on the *PPP* protocol is specific to the current implementation. Therefore, the results can be improved by improving the implementation. Some limitations in the implementation result in issues such as long experiment execution times. These issues are not part of the abstract ideas we have introduced, rather they are tied with the *PPP* implementation we currently have.

5.9 Summary

In this chapter, we introduced the partial panorama prediction (PPP) protocol. We specified the objectives of the protocol as to provide mobile clients with rich navigation schemes and maintain interactive response times. *PPP* protocol relies on predicting next user movements and streaming them in advance to the client. This is quite challenging having complex navigation options available to the user. The user may travel in any direction (without changing altitude). We introduced *partial panoramas* which are a subset of a full panorama. They save valuable bandwidth and speed up the system by streaming only parts of the panorama predicted to be used by the user. A minimum rotation coverage is guaranteed on the client, meaning that up to a certain amount of

rotations are pre-streamed to the client. When close to exceed the coverage, the server sends new parts of the panorama to maintain the coverage.

We also introduced the *strafing* idea and saw how it can generate the image at the new position mostly from the current image. This avoids sending the new image upon request and enables the client to use the current image to generate most of the new image. Strafing need pixels depths since it uses 3D transformations to transform the pixels. The idea of the strafe movement was generalized to any movement in any direction by strafing from a *key partial panorama.* A key partial panorama is placed in a position close to the user's current position and maintains a certain rotation coverage. Depending on this coverage amount, this key panorama can be rotated and used as a strafe source to generate images at close positions.

Combining the ideas, of partial panoramas, minimum rotation coverage and key partial panoramas, *PPP* is able to predict and pre-stream data needed to generate the next view or at least most of it. This results in a major achievement over traditional ideas where data is only sent upon request. *PPP* is always ahead of the user and it adapts to user movement patterns by changing its parameters. We then showed our implementation of *PPP* . We discussed the partial panoramas algorithm is details as well as depth compression/decompression.

In the next chapter, we evaluate the performance of *PPP* against existing ideas categorized either as *IR* or *POD* protocols.

Chapter 6

Performance Evaluation

At this point, we have developed a new image-based streaming protocol; *PPP* , implemented it and integrated it into our virtual environment streaming system. It is now time to evaluate its performance against other existing protocols. Section 1 of this chapter describes the hardware used for the rendering server and the mobile clients as well as the software used to implement the protocols and the virtual environment. Section 2 then explains what scenarios we use in our experiments and how we generate these scenarios. Section 3 presents a special prototype system that we used to evaluate the protocols' performance. Section 4 shows and analyzes our experiments results for different sets of experiments. These experiments were conducted on our implementation of the protocols explained earlier in this thesis, therefore results apply to the implementation in specific and can be improved by improving the implementation as we stated in chapter 5.

6.1 Experiments Hardware and Software Specification

The server machine has the following specifications:

- 2.8 GHz AMD Opteron processor.
- 2GB of RAM.
- Dual GPU Nvidia Quadro FX4500 X2 with 1GB of DDR3 RAM graphics card.

The mobile client is an HP iPAQ hx2790 PDA equipped with the following:

- Intel Xscale processor clocked at 624 MHz.
- 64MB of RAM and 192MB of ROM.
- 802.11b compliant radio interface.
- Windows Mobile 5.0 [58].

Regarding the network topology used in the experiments, the server is connected to a 802.11b compliant wireless access point, and the mobile devices (PDA) connect to the server through the wireless access point. No data encryption is used and there is no cross traffic in the testbed network.

The server software is fully written in C# 3.5 [59] and the virtual environment renderer is implemented in managed Microsoft DirectX 9.0c (MDX) [60]. The client software is implemented in C# targeting .net compact framework 3.5 [62] and the panorama renderer uses Windows Mobile DirectX [61], which is a limited subset of DirectX.

Figure 6.1 shows two screen shots of the system running. The first image shows six PDAs connected to the server and the second image gives a closer look to one of the PDAs.

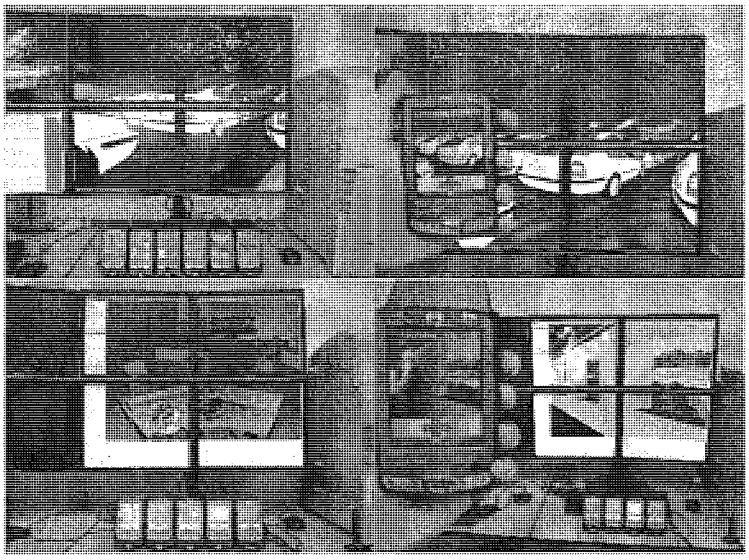


Figure 6.1: Virtual environment streaming system screen shots.

Figure 6.2 shows screen shots of the client application that runs on the PDA. The user selects a protocol and signs in and then can navigate the virtual environment using the selected protocol. The system supports multiple simultaneously connected clients each using a different streaming protocol.

6.2 Experiments Scenarios and Metrics

The streaming protocols need to be tested under many scenarios to truly evaluate their performance. Scenarios have many factors that control them. Some of these factors are:

- **Number of clients:** the number of clients online and navigating the virtual environment affects the performance of the protocols. The more clients connected, the more load is on the network and the server. This factor is used to show how each protocol scales with larger number of clients.

- **Client Navigation Paths:** navigation routes taken by each client, their size, movement type frequencies, and consistency, greatly affect the performance of

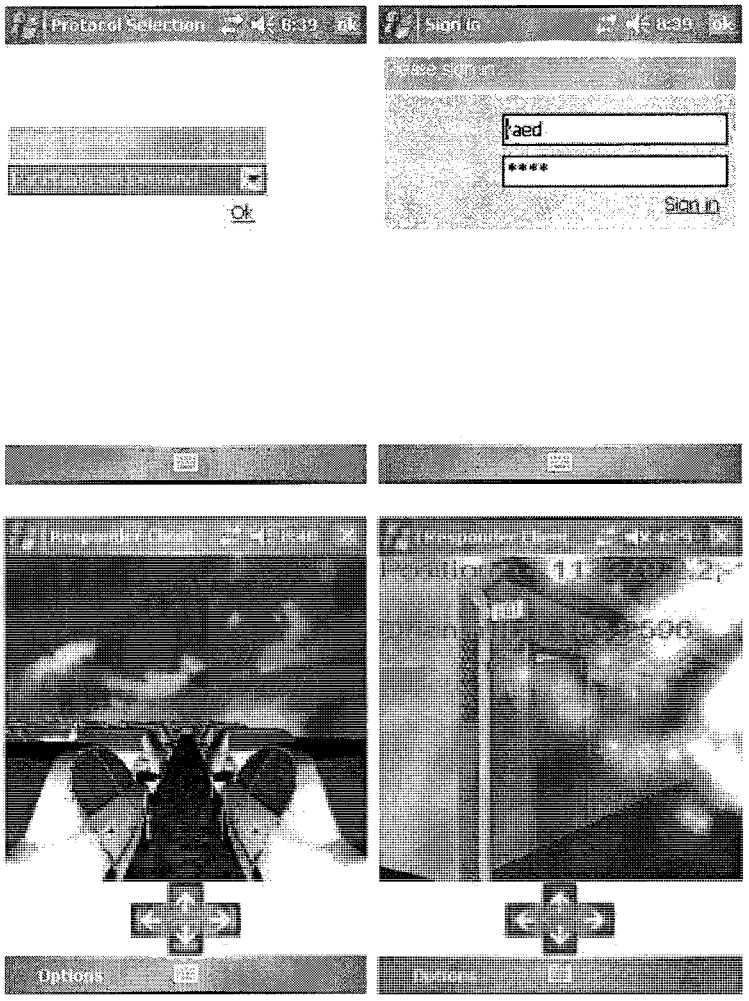


Figure 6.2: Client application screen shots.

streaming protocols.

- **Virtual Environment Complexity:** the complexity of the virtual environment affects the processing load on the server and therefore might affect the overall performance of the streaming protocols.

- **Specific protocol parameters:** some streaming protocols have parameters that control their behavior. For example, *PPP* can be set to different key panorama rotation coverages.

In our experiments, each client navigates according to a *movement script.* A movement script is a text file that specifies the steps the user takes within the virtual environment. The following commands show a small piece of a movement script:

$Forward, Forward, Forward, Rotation(5:0), Rotation(0:-2), Rotation(3,3),$
$Backward, LeftStrafe, Rotation(6:6), Rotation(13:0), Rotation(0:5),$
$RightStrafe, \ldots$

This script moves the user 3 steps forward, then rotates 5 degrees to the right, rotates 2 degrees up and so forth.

A movement script generator software was developed to automatically generate movement scripts to be used in the experiments. A screen shot of this generator is shown in Figure 6.3

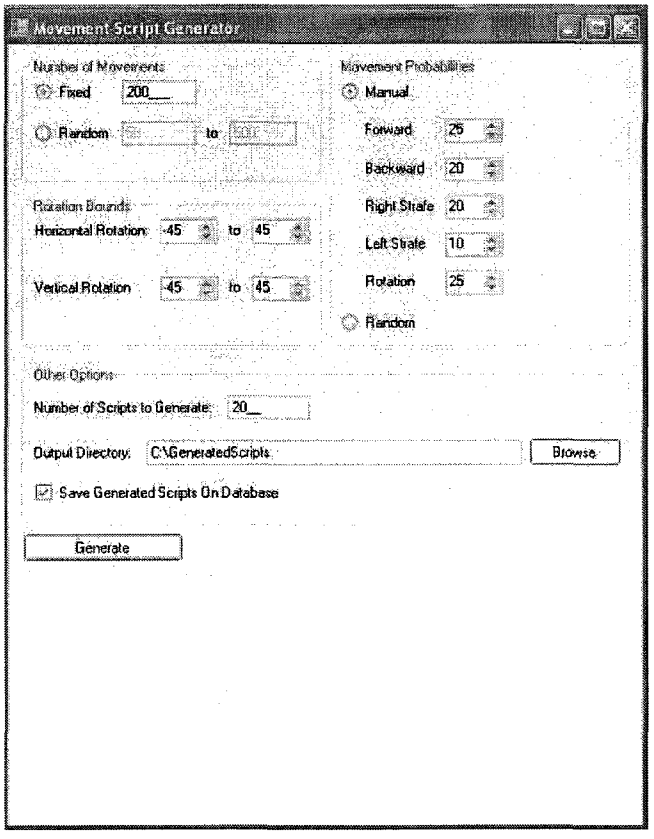


Figure 6.3: Movement script generator software.

The following criteria define the path to be generated:

- **Number of movements:** number of movements the user will move. Can be set to a fixed number or a random number in a range.
- **Movement probabilities:** the probabilities of each movement type, such as: forward, backward, rotation, etc., can be manually set or randomized.
- **Rotation bounds:** the range in degrees that a client can rotate by in both horizontal and vertical directions. The user is free to rotate any where in the environment, these bounds only limit the amount of rotation per script command.

The number of scripts to generate according to previous criteria can be set as well as the output folder of these scripts. An option to save the generated script(s) to a database will be described in later sections where we define the experiments framework we use.

In our experiments, we evaluate the protocols according to different metrics. we define these metrics as follows:

- **Waiting time:** the time difference measured between a movement request issued at the client and the time the client actually displays the scene it requested.
- **Network traffic:** the amount of data transmitted on the network to fulfill a movement request.
- **Local cache hit:** a percentage that indicates if the requested view was already stored on the client. A 0% cache hit means that no data at all was on the client and the server had to stream the whole new view. A 100% cache hit means that the client had the requested view in advance and did not have to request it from the server. A value between 0% and 100% indicates that a portion of the view was on the client while the rest was transmitted from the server.
- **Bandwidth utilization:** the amount of data transmitted every second. Represents the protocols usage of the network bandwidth.

6.3 Experiments Framework

A special prototype system was developed in order to evaluate the performance of the proposed streaming protocols. A *performance test server* holds the server protocols and accepts connections from *performance test clients* deployed on the PDAs.

A database is used to store experiment runs and results. This database also stores movement scripts generated using the tool we described in the pervious section, and their different attributes.

Initially, the performance test server starts up and listens on a special port for incoming connection requests from test clients. Test clients sign in sending their field of view and requested image size (Figure 6.4). Thereafter, for each client, the streaming protocol can be selected. The performance testing server is shown in Figure 6.5(a). The server can configure a client to run more than a session. This means the client can host multiple sessions to simulate multiple clients. This feature is developed to test large number of clients, however, it was not used in this thesis experiments. All clients in our experiments only run a single session.

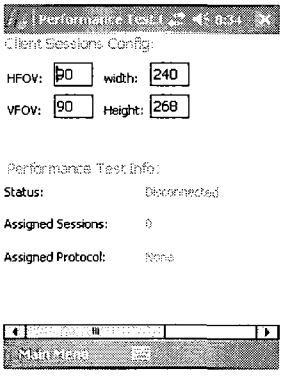


Figure 6.4: Performance test client.

For each client session, a movement script that defines the user's movement is assigned. A movement script can be randomly selected from the database or selected from a file or selected from the database according to some conditions, such as: number of movements and movement probabilities whether fixed or in a certain range. Configuring

a session's movement script is shown in Figure 6.5(b).

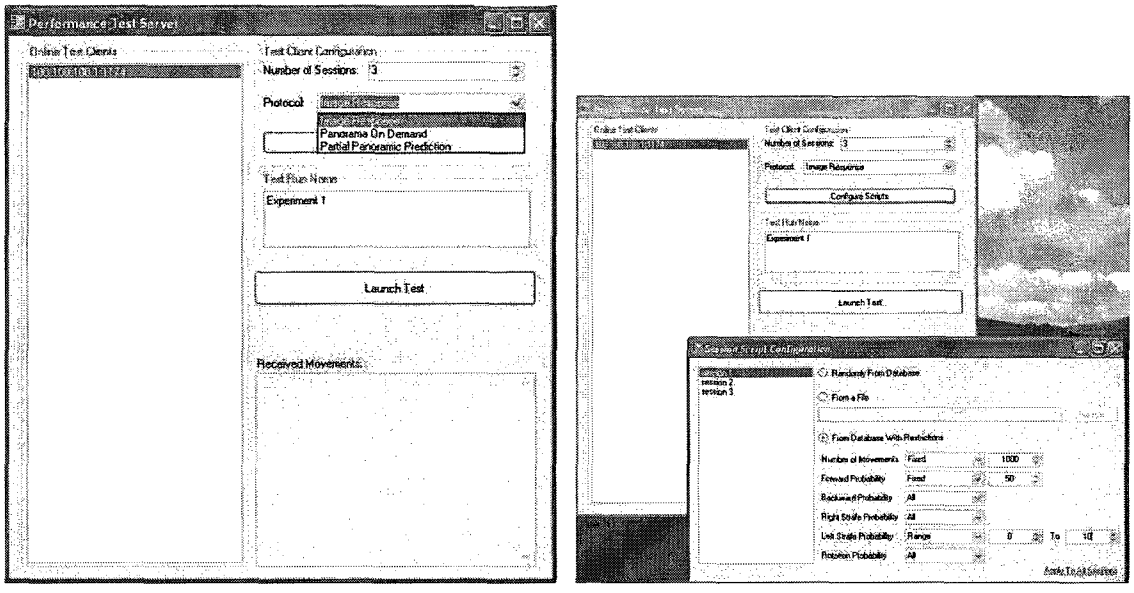


Figure 6.5: Performance test server.

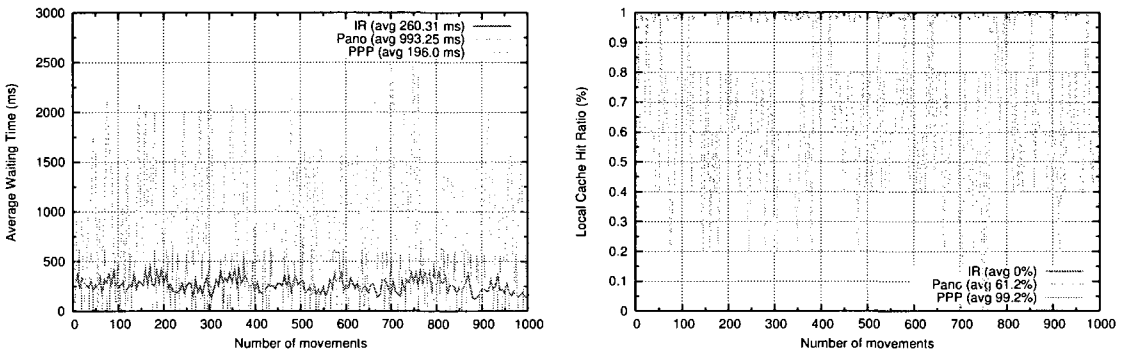
After setting the protocols and configuring sessions for clients, the performance test commences. The server sends each client its assigned streaming protocol, number of sessions and movement script. The performance test client then loads the client protocol and starts navigating the environment. The client device keeps the server updated with the current viewpoint of the user. Depending on the position of the user and the data cached at the client, the client may request panorama segments and/or the pixel difference when using the strafe algorithm. The server keeps track of the movement of the virtual user and sends the required data accordingly. All the metrics mentioned in the previous section are calculated and saved into the experiments database.

6.4 Performance Evaluation Results

In this section, we show the experiments we conducted. For each experiment, we specify the scenario used, show the results and analyze the results and justify them. The experiments measure implementations of three streaming protocols. We define them as *image response* (IR), *panorama on demand* (Pano) and *partial panorama prediction* (PPP). In the IR approach, the client keeps requesting images as the user moves through the virtual environment, i.e., one movement one request. Similarly to IR, in the Pano approach the client sends requests for each single movement of the user, however, instead of images the server sends full panoramas. The user is able to rotate within the panorama, but it requires another panorama as the user moves to another position. The former two protocols are implemented by most existing solutions discussed in chapter 2. PPP sends a partial panorama at the user's position as well as a number of partial key panoramas to generate new views the user might request. We compare these protocols using the same platforms, softwares and methods. We run each experiment ten times and average them. A 95% confidence interval was used. A summary of the parameters of our experiments is shown in table 6.1.

Table 6.1: **Experiment parameters.**

Parameter	**Value**
Image Size	88.43 KB
Image Resolution	240 × 268
Image Format	PNG
Horizontal FOV	90°
Vertical FOV	90°
Number of Clients	1 – 8
Network Medium	802.11b
Network Bandwidth	11*mbps*
Client Number of Movements	1000



(a) Average waiting time **(b) Client's local cache hit ratio**

Figure 6.6: Average waiting time and local cache hit ratio for a random path.

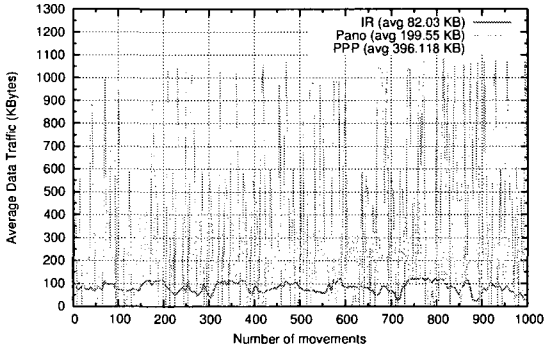


Figure 6.7: Average data traffic for a random path.

6.4.1 Random Navigation Path Experiments

In the first set of experiments, we evaluate the protocols performance as a function of the number of movements. To this end, we have generated a script composed of 1000 random movements that represents the users navigation within the virtual environment. As depicted in Figure 6.6(a), the average waiting time of our proposed protocol outperformed the other protocols evaluated. This is due the following characteristics of our scheme: firstly, we employ partial instead of full panoramas, which explains the advantage of our approach compared to the panorama on demand; secondly, the data required for the next

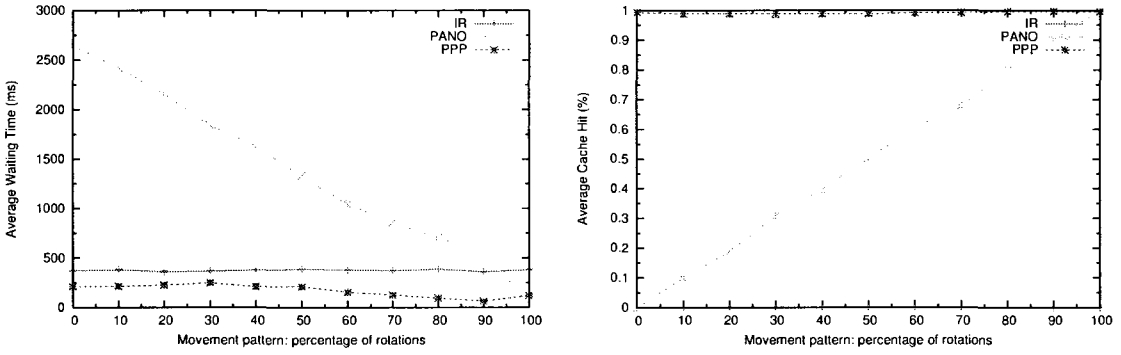
movement is always pre-streamed to the client, this includes the panorama segment that represents the current viewpoint plus key partial panoramas used to generate new views based on the most probable future users movements. In the case of a user rotation, the current partial panorama guarantees that the next view is already cached at the client. Therefore, our scheme outperforms the IR approach since an IR client requests a new image for every single users movement. In the case of a change in the users position, e.g., forward, backward, and strafe, the PPP protocol will either use a key partial panorama when the user moves towards its position, or strafe from the closest key partial panorama. Therefore, the client will have most of the new view, if not all of it, already available in its cache. When the next view is partially available at the client, the missing parts are streamed from the server. These missing parts are significantly smaller than an entire image or panorama. The long waiting time observed in the panorama on demand approach is caused by the streaming of entire panoramas upon each user movement. The high local cache hit percentages of PPP, 99.2% in average, explains the advantage of our protocol, as shown in Figure 6.6(b). The IR approach does not use any cache for next user movements, thereby its cache hit is always 0%. The panorama on demand, on the other hand, streams full panoramas and the new views are rendered locally when the user rotates. However, when the user moves, a new full panorama is streamed, which results in a low cache hit ratio.

The averages of the data traffic generated by the protocols are presented in Figure 6.7. As expected, among the three protocols, IR is the one that uses less bandwidth on average. This is due to the fact that IR only sends the required image for the current view, whereas panorama on demand streams a full panorama when the user moves, and PPP streams multiple partial panoramas, including compressed depth values for the current position and key partial panoramas. The high bandwidth usage of PPP enables the scheme to achieve better delay performance than the evaluated protocols since most of the imagery data is streamed and cached in advance. This way, the client has all the data available locally before the user moves to a new postion.The high bandwidth

of PPP is majorally caused by sending depth values. Although we use a compression scheme that sends an depth index list instead of raw depths, depth values still resemble most of what is being sent. As future work, depth sizes can be reduced by using an in-memory compression scheme to compress depth's float values.

6.4.2 Rotation Weighted Path Experiments

In the second set of experiments, we evaluate the protocols under movement scripts with different percentages of rotations.



(a) Average waiting time **(b) Client's local cache hit ratio**

Figure 6.8: Average waiting time and local cache hit ratio for a rotation weighted path.

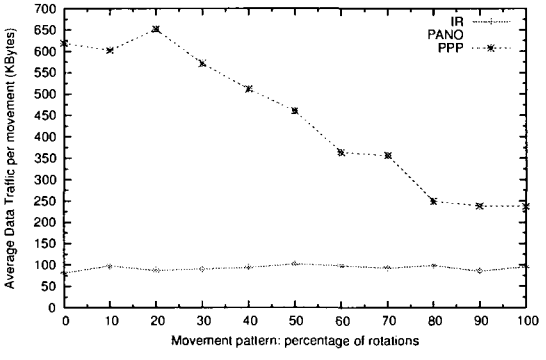


Figure 6.9: Average data traffic for a rotation weighted path.

The reason of these experiments is that the percentage of rotations greatly affects the performance of the protocols. For instance, panorama on demand performs poorly when the amount of rotations is small, as depicted in Figures 6.8(a) and 6.8(b). Notice in Figure 6.8(b) that the cache hit ratio is directly proportional to the percentage of rotations in the panorama on demand protocol. This leads to much smaller waiting times when the user's navigation path is mostly made of rotations. This is clearly visible in Figure 6.8(a); the average waiting time for a path with no rotations is more that 2.5 seconds where it is a little bit over 0.5 seconds when 90% of the path is made of rotations.

The amount of rotations has no impact on the IR protocol since it does not distinguish between a rotation and a movement. The PPP protocol shows stable behavior mostly because it pre-streams imagery data needed for any future movements. Average waiting times falls down a bit as the percentage of rotations increases. Upon a rotation, the rendering server does not have to render a new panorama, where as when moving, the server renders a new panorama plus a set of key panoramas. This rendering actually takes a considerable time and will be shown to greatly affect the performance of our PPP protocol implementation in later experiments as the number of online client increases and more rendering load is put on the server. The consistent cache hit ratio of PPP in Figure 6.8(b) supports this hypothesis. It is evident that the problem is not in the prediction, but rather is in the rendering routine.

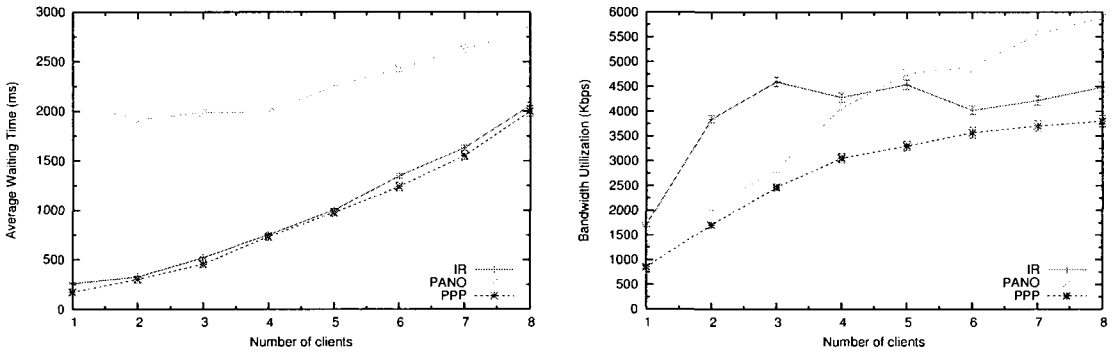
The current implementation of the rendering engine is not optimized. This has negatively affected the overall results of the protocol. It still outperforms the other protocols but not as much as it should. Rendering efficiency can be significantly improved by applying appropriate DirectX techniques to burst PPP results. This is left as future work.

As depicted in Figure 6.9, the average amount of data traffic for the panorama on demand drops dramatically as the percentage of rotations increase. This is directly tied with improved waiting times and local cache hits. As in the first set of experiments, PPP consumes more bandwidth that the other protocols. The amount of data streamed

generally decreases as the rotations in the path increase since less depth values are streamed as fewer key partial panoramas are sent.

6.4.3 Multiple Clients Experiments

In this set of experiments, we measure the protocols performance with multiple clients connected simultaneously. We conducted the experiments on up to eight PDAs connected and navigating the virtual environment at the same time.



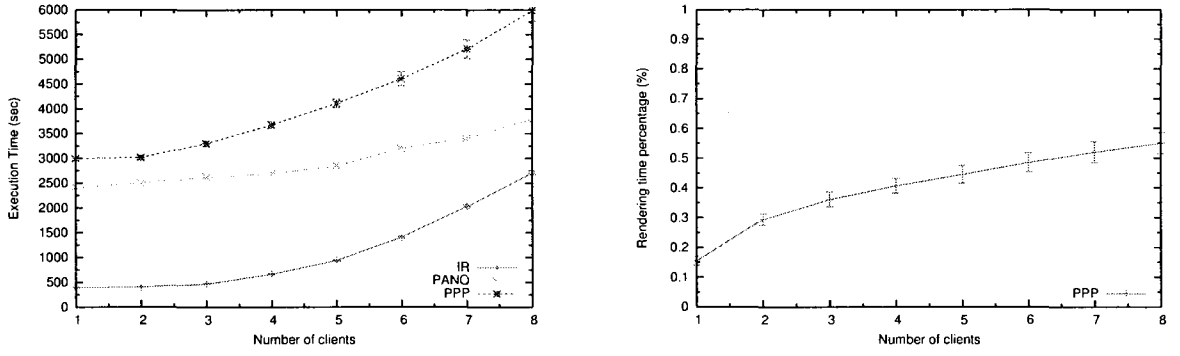
(a) Average waiting time **(b) Bandwidth utilization**

Figure 6.10: Average waiting time and bandwidth utilization vs number of clients.

As shown in Figure 6.10(a); the waiting time for all protocols increase as the number of clients increase. Panorama on demand is always worse than IR and PPP. PPP maintains a slightly faster response time than IR. This is explained by the rendering bottleneck created in the server due to rendering a lot of panoramas. This issue can be fixed by optimizing the cubic panorama rendering technique and using multi-threaded rendering routines in future versions of the software. This fact is graphed in Figure 6.11(b). The rendering time percentage of the the total response time increases largely as the number of clients increase. It starts at 15% for one client and nearly doubles for two clients, and finally escalates to more than 50% of the total time.

The graph in 6.10(b) shows the bandwidth consumption of the streaming protocols.

Although PPP protocol sends more information, it appears that it has the least bandwidth utilization. This is because the bandwidth is measured by dividing the amount of transmitted data over the total experiment time shown in Figure 6.11(a). PPP experiment time takes much longer than the rest of the protocols. This is an implementation limitation done to synchronize sending all of the keys panoramas requested for a single movement before processing the next movement. In a real application of the protocol a roll back policy has to applied to the rendering server i.e. while generating the key panoramas for a movement, the server received another movement, it should cancel what its doing and service the new movement. This is an advanced topic that emerged during the implementation of our proposed protocol and that needs further research. This problem will be addressed in the near future work.



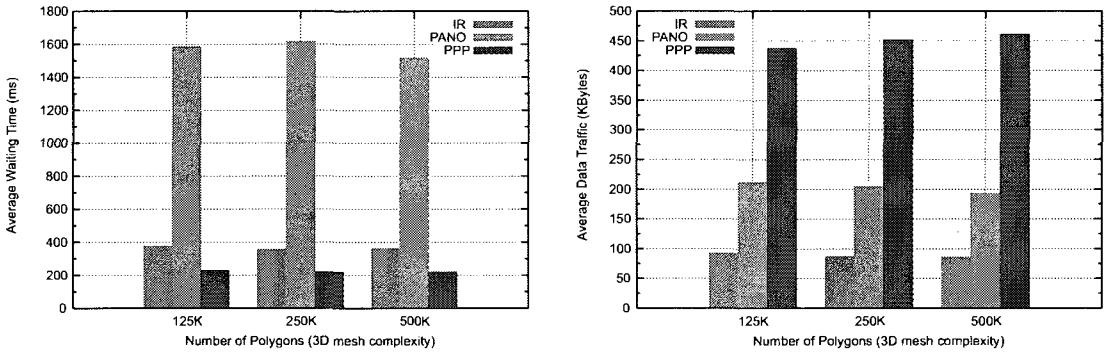
(a) Total execution time **(b) Rendering time percentage of total time**

Figure 6.11: Execution time and rendering time percentage vs number of clients.

Overall, PPP achieved better results than the other two protocols. Data availability on the client is higher and user's next movements prediction enables highly interactive user experience. Once the rendering bottleneck in our implementation is solved, and depth compression is enhanced; PPP is expected to achieve much better results than the current results of the current prototype implementation.

6.4.4 Virtual Environment Complexity Experiments

The performance of image-based streaming protocols is evaluated against virtual environments with different complexity. We have modeled three virtual environments with different complexities, i.e, number of polygons. As can be seen in Figures 6.12 and 6.13, the performance of IBR streaming protocols is not affected by the complexity of the scene. This enables the rendering of complex virtual environments at interactive frame rates on resource constrained mobile devices and proves that IBR techniques isolate the complexity of the virtual environment since it delegates this heavy task to a powerful rendering computer.



(a) Average waiting time **(b) Average data traffic**

Figure 6.12: Average waiting time and data traffic for different VE complexities.

A remark on Figure 6.13 is that bandwidth utilization for PPP and Pano are less than IR although they send more data across the network because of the time wasted on rendering cubic panoramas which is not present in IR. This processing bottleneck can be solved as we mentioned earlier i.e. optimizing panorama rendering, and by using a more powerful server machine designed specifically to render virtual environments efficiently.

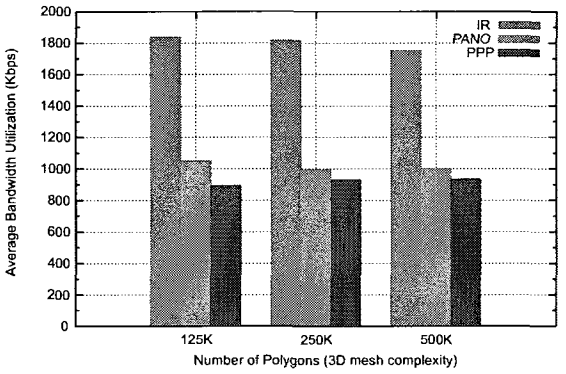


Figure 6.13: Average bandwidth utilization for different VE complexities.

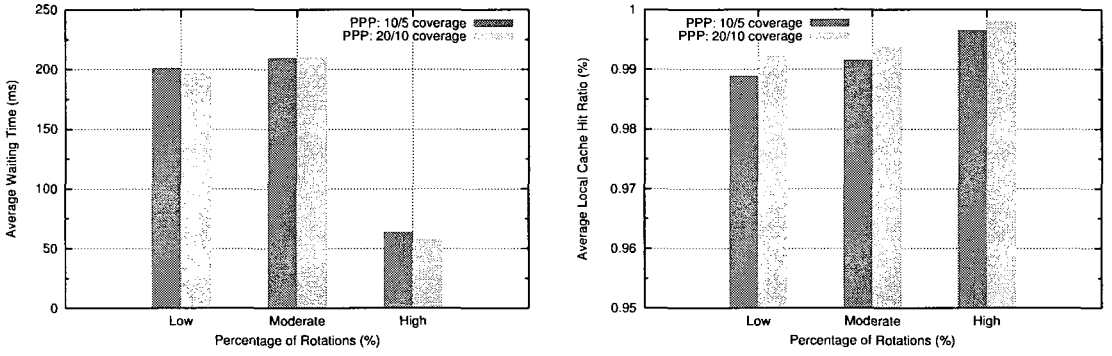
6.4.5 Rotation Coverage Experiments

In the last set of experiments, we compare two different versions of PPP protocol. Adaptation to user movement is not currently implemented. Therefore, protocol parameters such as the amount of rotations covered by a key partial panorama are fixed. The first version of PPP uses 10 degree horizontal and 5 degree vertical rotation coverage for key partial panoramas, where as the second version uses 20 degree horizontal and 10 degree vertical rotation coverage.

These two versions are compared against paths with different rotation intensities. We used paths with 10%, 50% and 90% of rotations, considered as low, moderate and high rotation paths. Figures 6.14 and 6.15 show the performance results of both PPP versions.

It is noted in Figure 6.14(a) that the average waiting times for both versions is nearly identical and that for the high rotation path, both versions had much shorter waiting times than in moderate and low rotation paths. This further supports the argument that rendering cubic panoramas consumes most time. In the high rotation path, fewer panoramas had to be rendered since there are less movements. Therefore, waiting times were much smaller.

The local cache hits for the higher coverage version of PPP is slightly higher than the lower coverage version (Figure 6.14(b)). This is because that the amount of degrees



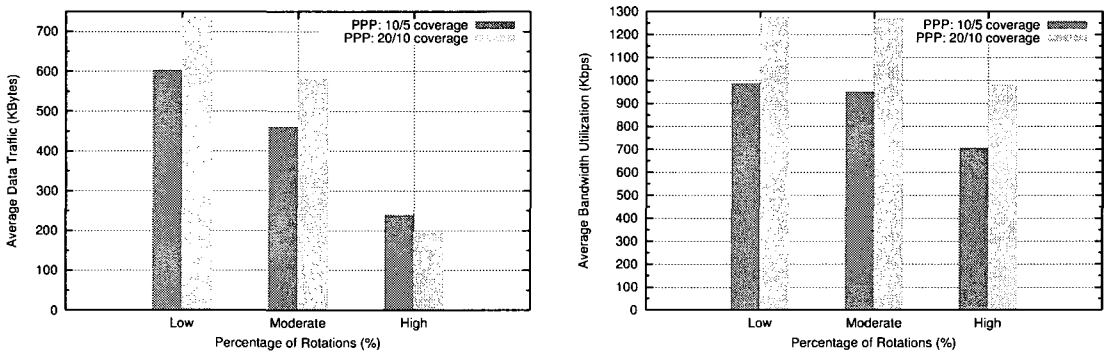
(a) **(b)**

Figure 6.14: Average waiting time and local cache hits for different *PPP* coverages.

the user rotates by was not large in most of the rotations and therefore needed views were almost always available on the client. Another reason is that we detect when the user is close to exceed the rotation coverage and send new segments of the panorama to maintain the same coverage.

As expected, data traffic for the higher coverage version is larger for low and moderate rotation paths. This is obvious since we stream more panorama segments that were not used in most of the cases. However, for the high rotation path, the numbers are inverted; the higher coverage version actually streams less data. This is explained by the fact that when coverage is small and rotations are high, more key panoramas - covering less rotations - are streamed to the client to predict possible movements. When the coverage is high and the rotations are high too, less key panoramas are streamed to cover possible movements since these key panoramas cover more rotations.

As depicted in Figure 6.15(b), the bandwidth utilization of the higher coverage version is higher. This is because less panoramas are needed to be rendered and more panorama segments are sent which mean more traffic and less rendering times. This results in higher bandwidth utilization rates.



(a) (b)

Figure 6.15: Average data traffic and bandwidth utilization for different *PPP* coverages.

6.5 Summary

We evaluated the performance of the streaming protocols developed in this thesis. We specified the hardware configuration used in testing as well as the software technologies we used to implement our streaming system and its protocols. Then, we defined the experiments scenarios and classified their factors such as: navigation paths and number of clients participating in the test. Further, we specified the metrics we use in our evaluation, such as: waiting times, data traffic and local cache hit ratio. We then explained the performance testing framework we developed and showed how test clients sign in to the test server and how the test server assigns protocols to these clients.

The results of the experiments where then shown in different sets representing different scenarios. Experiments were conducted for: random movement and rotation weighted paths, single and multiple clients, different VE complexity and different PPP protocol parameters.

We have seen that PPP outperforms the other two protocol classes, IR and Pano due to its advanced features that allow future data to available on the client before needing it. Although current PPP implementation performs well, it can be considerably improved by solving current implementation limitations. Rendering cubic panoramas in the current

implementation takes significant amount of time especially when the number of clients increases. It reaches to more than 50% of the total waiting time when eight clients are connected simultaneously. This issue has degraded PPP performance and if solved, it is expected that PPP results will improve. Another issue noted is that PPP streams data more than the other two protocols. This was diagnosed as sending large depth volumes. In the future, an in-memory compression in addition to the current depth compression scheme should lower the data traffic caused by PPP.

Chapter 7

Conclusions

In this thesis, our goal was to enable the exploration of complex 3D virtual environments on mobile devices and provide an interactive experience to mobile device users in terms of movement flexibility and system response times, effectively turning many interesting 3D environment applications on light mobile devices into reality.

Among the many existing techniques for networked virtual environments, we used image-based rendering *(IBR)*. In this technique, a server renders the views and sends them as images to clients. The heavy rendering task is delegated to the more powerful server, alleviating the load on the client, which only displays received images according to an IBR algorithm. Therefore the complexity of the environment is neutralized, since the client only receives images. Only the size of the image and its contents affects the performance on the client. There are various algorithms for reconstructing views from the received images on the client. These algorithms are all based on the plenoptic function [16] and implement a subset of it. They try to construct complete views from discrete samples. We used panoramas to display all possible views at a certain position in the virtual environment. As seen in the related work chapter, current IBR approaches that try to solve the problem of efficiently displaying 3D virtual environments on mobile devices suffer from being reactive to users and providing limited movement schemes.

We began solving these problems by developing a virtual environment streaming

system. This system was carefully designed to enable multiple simultaneously connected mobile clients to navigate a 3D environment, each client using a different streaming protocol. The system includes components that facilitate network communication and messaging, as well as the 3D virtual environment, session management and protocol placeholders. All streaming protocols are seamlessly plugged into the system.

Having all what a streaming protocol needs, we categorized existing IBR solutions under either image response *(IR)* or panorama on demand *(POD)* protocols. *IR* renders the requested view into an image and sends it to the client once the client informs the server of an update. *POD* renders a complete panorama that covers looking in all directions from a position and sends the panorama to the client. The client uses a panorama rendering mechanism and renders the received panorama. We stated that *IR* protocol is completely reactive and does not predict or study user movements at all. All movements are considered same and each time the client moves, it has to suffer network latencies and rendering delays until it receives the requested image. *POD* tries to predict user movements by sending a complete panorama for every position the user travels to. This helps if the user rotates at his current position. In this case the panorama present on the client generates the new view for the clients new looking direction. This protocol performs badly if the user changes position a lot. The size of a panorama is usually much larger than the size of the viewable segment the user sees at a time, therefore sending complete panoramas for every position wastes a lot of time and bandwidth and most of the time a major part of the panorama is not even used by the client.

These problems motivated us to come up with a new idea. An idea that predicts next user movements more smartly and does that for a complex navigation pattern. The key to achieving convincing performance is caching. Making the expectedly requested views available ahead of time on the client dramatically improves system performance since the client does not have to wait for network delays and server rendering times once it needs a new view. This is the main idea that the new idea achieves.

The new protocol, partial panorama prediction *(PPP)*; uses many innovated ideas to

accomplish that. First of all, we introduced *partial panoramas*; a subset of a complete panorama that covers a certain amount of horizontal and vertical rotations instead of the whole 360 degree view. By manipulating the amount of coverage of a partial panorama we are predicting that the user will only use covered rotations. Therefore we save streaming unnecessary parts of the panorama. A rotation coverage is guaranteed on the client by the protocol, if the user is close to exceed the coverage of the partial panorama, the system sends new parts of the panorama to maintain the minimal rotation coverage. This means certain parts of the panorama are only streamed once the user is expected to need them. This is a big difference than sending a complete panorama every time regardless of the user's need as *POD* protocol does. This technique effectively predicts future user rotations. The system keeps track of user's rotations and changes the amount of coverage depending on how often the user rotates at each position it visits in the virtual world.

Prediction of movements was accomplished by a set of techniques. Firstly, we described a special movement type called *strafing*. A strafe is a horizontal movement that does not involve any rotation. Using 3D mathematics, we developed a method that generates the new view after a strafe from the client's current view. This method produces most of the new image using 3D transformations. Therefore pixels depths are needed for this method to work. In the example we provided in section 5.5, at least 95% of the new image is generated by the method. This strafe algorithm saves a lot of waiting time, since the client has only to wait for 5% or less of the new image to arrive instead of waiting for a whole new image. Including the depth overhead in the same example, around 34% saving in bandwidth is also achieved.

To predict user's next movements, we transform any movement in any direction into a strafe movement for which we have developed an efficient method that utilizes cached information on the client. *Key partial panoramas* are partial panoramas that are carefully placed to generate movements in certain directions. The prediction of next user movements is now transformed into predicting which key partial panoramas to send to the client in advance. As shown in chapter 5, each key partial panorama covers movements in

directions up to a specific amount of degrees away from the key panorama's origin angle. When the user rotates and is close to exceed the coverage of a key partial panorama, the system sends a new key partial panorama that covers new directions uncovered by the old key partial panorama. Thereby, only key partial panoramas closest to user's direction are streamed to the client and only if the client needs them. *PPP* ensures that either the newly requested scene is available on the client in full such as the case of a rotation or at least most of it as in the case of strafing from a key partial panorama. *PPP* differs from the other two protocols in a very important aspect, *adaptiveness.* Depending on the movement pattern of the user, *PPP* changes its parameters and might decide to omit sending data necessary to cover certain movement types that it does not expect the user to undertake. Other protocols only respond to user movements with out understanding them.

In our implementation of *PPP* , we used cubic panoramas which we found the simplest and the most efficient to render. We developed a new axis system for representing partial cubic panoramas and an algorithm that calculates new face pieces to send for each rotation and keeps track of sent pieces. In addition, we developed a depth compression algorithm to minimize depth overhead and boost performance. The compression algorithm relies on sending a depth list that contains unique depth values, and encoding depth list position in every pixel, and therefore saves sending same depth values more than once.

The results obtained from conducting an extensive set of experiments that cover various scenarios show that *PPP* protocol outperforms *IR* and *POD* protocols, mainly due to its caching and prediction mechanisms. Results also show that the complexity of the 3D virtual environment almost does not impact our system's performance and proves that IBR enable clients to display compelling 3D scenes similarly to simple ones. Results can be more improved by addressing the issues that the implementation has. Many aspects have not been optimized. Rendering cubic panoramas is done in a simple way and is currently considered slow. With careful techniques related to the 3D library used,

rendering times can be massively improved. More advanced algorithms for computing partial panoramas can be developed in the future. Another formats of panoramas can be used if proved to be more efficient and easier to render than cubic panoramas. The current implementation does not include adaptiveness nor look ahead strategies. All the parameters are static at the moment. This feature is left for the future for more advanced research.

7.1 Future Work

For future work, current implementation issues and optimizations listed above should be considered. In addition, new techniques and ideas for streaming dynamic scenes should be researched and studied in depth.

The following points list optimizations and enhancements to the current ideas we have:

1. Panorama rendering optimization. If optimized, the system is expected to perform much better.

2. Developing better and more accurate partial panorama calculation methods. This depends on the type of the panorama used.

3. Solving Occlusion problem, by using splatting techniques for example.

4. Using in memory depth compression to further compress depth values and improve system's performance.

5. Implementing adaptation routine, by studying user movements and changing protocol parameters and setting movement type probabilities accordingly.

Our ideas can be further expanded to support even more complex user navigation such as leans and altitude change (jumps and crouches). Ideas for identifying dynamic

parts of a scene and streaming them side by side to static views have to be developed to provide a complete solution. Dynamic objects can be sent as small sub images for instance or can be sent as progressive meshes. Using the appropriate strategy to stream dynamic objects should be the main concern for future works for this thesis.

Appendix A

Demo Application

In this appendix, we present a demo application we developed that is built on top of the VE streaming system we have developed in this thesis. The VE streaming system provides generic services used to stream any virtual environment using the image-based protocols we developed and can be used in many applications.

The demo application is an *Emergency Preparedness* software used by fire fighting departments. The *target building*, i.e. the building which is monitored, is equipped with a wireless sensor network that measures readings such as smoke level and room temperature. A 3D virtual environment of the target building is used by the command center to represent the actual target building. Once the wireless sensor network detects a threat such as an increase in smoke levels, first responder units (fire fighters) are sent to the target building to deal with the situation. Each responder wears a head mounted display (HMD) or carries a PDA which is connected to the command center server software. All information necessary for the responder to accomplish his/her task is fed to his device.

The responder can see a virtual representation of the building as well as a map showing the building schematics. As he moves within the building, his position in the VE changes and new images are fed to him. Positions of his team mates as well as civilians are made available to each responder using information collected from the sensor network

and passed to the command center server. A virtual representation of threats are also displayed on his device to make him aware of the threat and its nature. For instance, invisible threats such as gas leaks are easily visualized within the target building VE. The command center on the other hand monitors all the deployed units and sends commands telling them what to do. For example, it can send the path to the nearest exit to each responder. Figure A.1 shows the command center running on a multi-screen server and first responder client application running on a PDA.

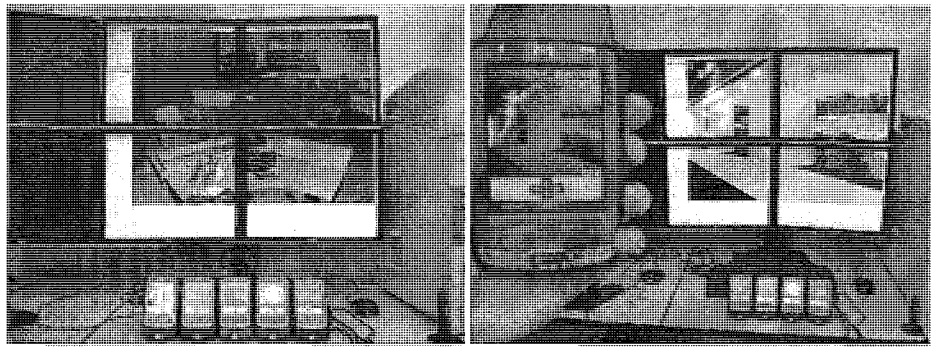


Figure A.1: Command center and first responder client software.

We used 3D Studio Max [81] to model the SITE building (School of Information Technology and Engineering) of the University of Ottawa. The command center server application is shown in Figure A.2.

We assume that a fire breaks out in the second floor of the SITE building, as shown in the left image of Figure A.3. The command center gets notified by the wireless sensor network and deploys first responder units to help civilians escape from the hazard. A first responder signs in to the system using his device. He is know added to the VE as shown in the right image of Figure A.3.

First responders will move to rescue the civilians and take them to the nearest exit as shown in Figures A.4 and A.5. The command center is updated with the position and status of all first responders, civilians and the threat.

Figure A.6 shows a first responder unit safely guiding a civilian out of the second floor as well as the responder's position on the map.

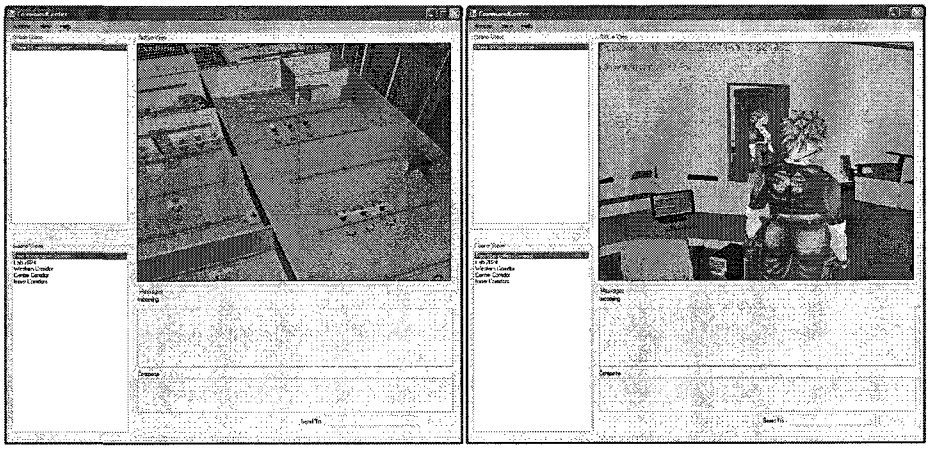


Figure A.2: VE streaming system. Left: Second floor view. Right: Paradise lab.

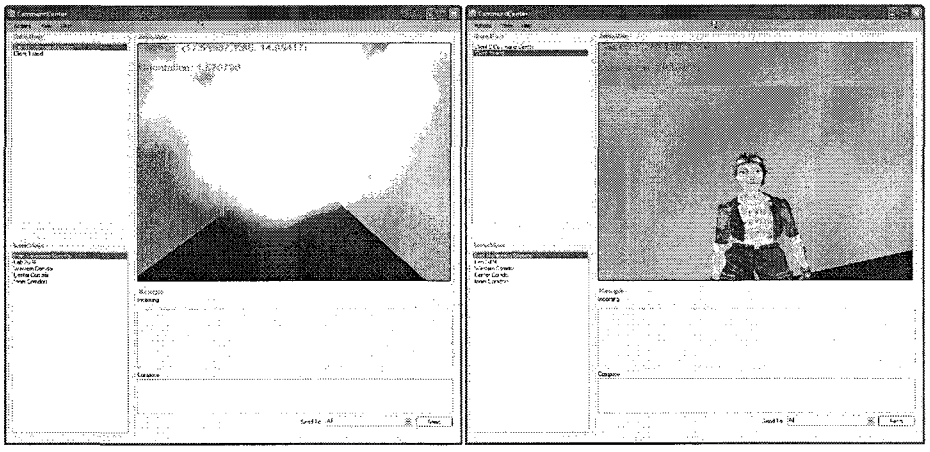


Figure A.3: Left: Fire. Right: First responder.

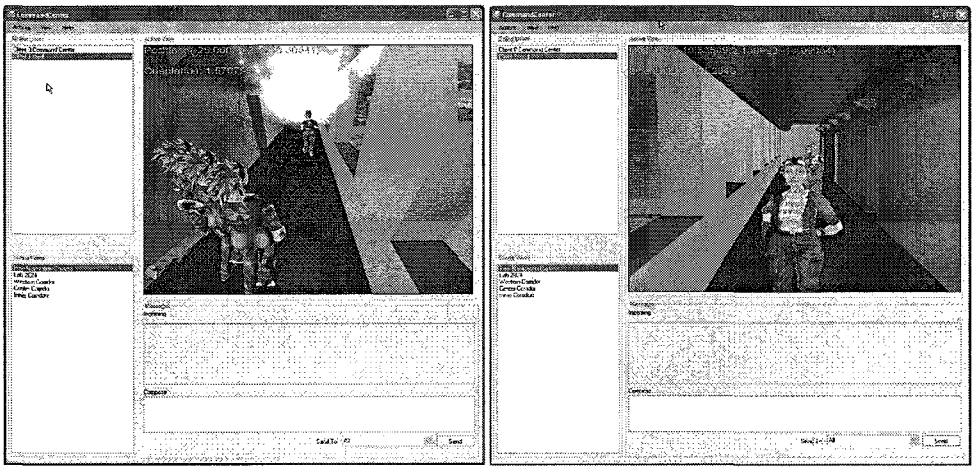


Figure A.4: Rescue process by first responder.

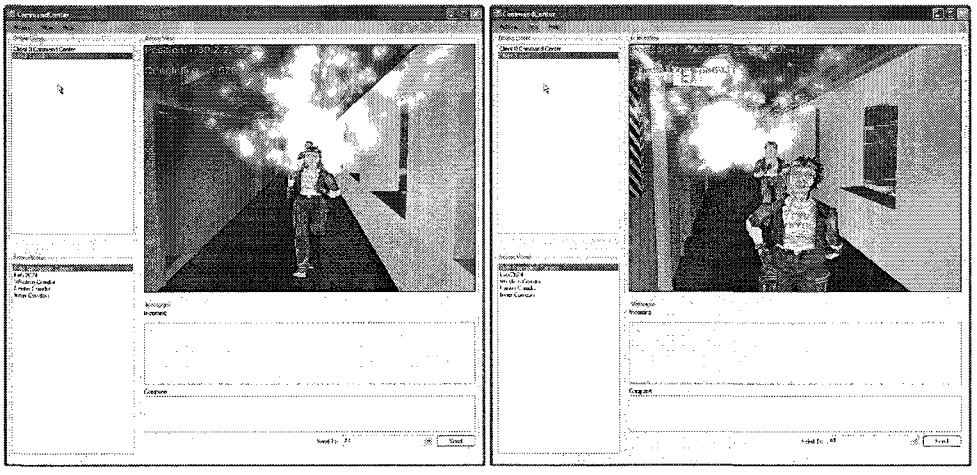


Figure A.5: Rescue process by first responder.

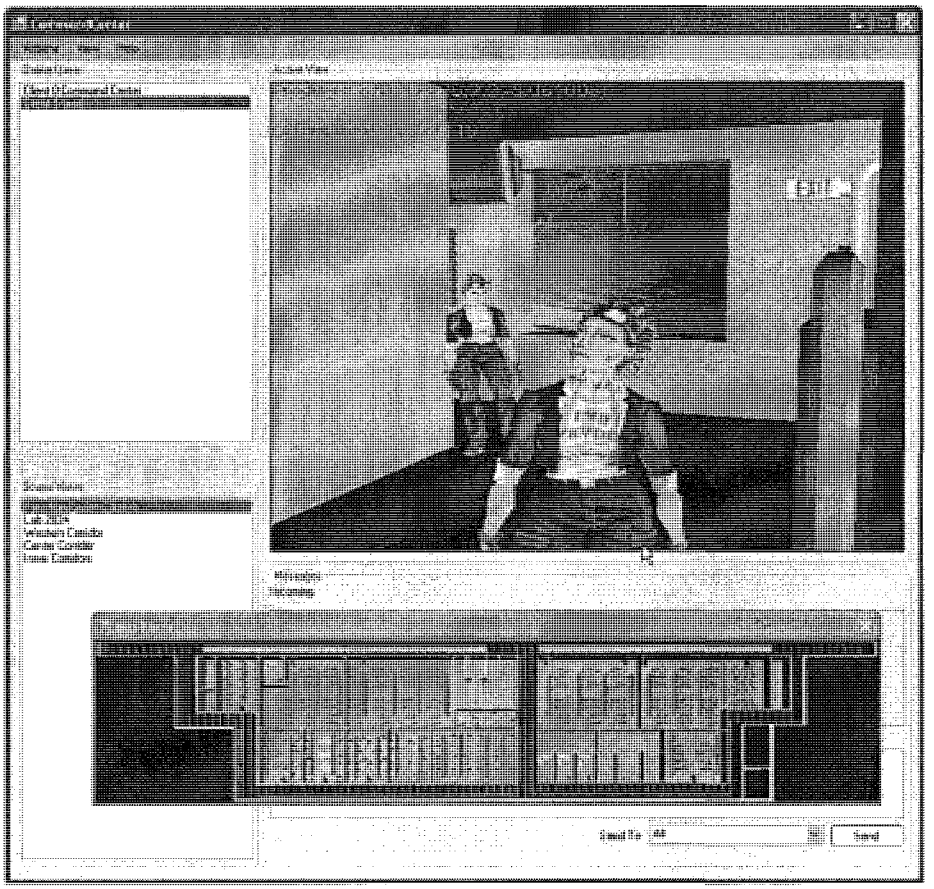


Figure A.6: Floor map.

Bibliography

[1] E. Lengyel. "Mathematics for 3D Game Programming and Computer Graphics" book of Algorithms for Wireless Networking and Mobile Computing". *Chapman & Hall/Crc Computer & Information Science.* Charles River Media, pp. 79-99, ISBN: 1-58450-037-9, 2002.

[2] A. Bonotti, L. Genovali, and L. Ricci. "DiVES: a distributed support for networked virtual environments." *In Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference* on Volume 1, pp. 107-112, 18-20 April 2006.

[3] R. Cavagna, C. Bouville, and J. Royan. "P2P Network for very large virtual environment." *In Proceedings of the ACM Symposium on Virtual Reality Software and Technology. VRST '06.* pp. 269-276, Limassol, Cyprus, November 2006.

[4] C. Joslin, I. S. Pandzic and N. M. Thalmann. "Trends in networked collaborative virtual environments." *Computer Communications.* Volume 26, Issue 5, pp. 430-437, March 2003.

[5] ID SOFTWARE, DOOM 3, Computer game (2006). http://www.idsoftware.com/games/doom/doom3/

[6] SimBim Development Team, GTR, Computer Racing Game (2006). http://www.simbin.se/

[7] H. Hoppe. "Progressive Meshes". *Proc. SIGGRAPH'96* , pp. 99-108, New Orleans, LA, USA, 1996.

[8] S. Kircher, and M. Garland. "Progressive multiresolution meshes for deforming surfaces." *In Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation*, Los Angeles, California, July 2005. SCA '05. ACM Press, New York, NY, pp. 191-200.

[9] J. Peng, and C. J. Kuo. "Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition." *Trans. Graph. 24*, pp. 609-616. July 2005.

[10] S. Yang, C. Lee, and C. J. Kuo."Optimized mesh and texture multiplexing for progressive textured model transmission." *In Proceedings of the 12th Annual ACM international Conference on Multimedia* , New York, NY, USA, October 2004. MULTIMEDIA '04. ACM Press, New York, NY, pp. 676-683.

[11] D. G. Aliaga, and A. A. Lastra. "Architectural walkthroughs using portal textures." *In R. Yagel and H. Hagen, editors, IEEE Visualization 97*, pp. 355-362, Nov. 1997.

[12] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. "Geopostors: a real-time geometry / impostor crowd rendering system." *In Proceedings of the 2005 Symposium on interactive 3D Graphics and Games*, Washington, District of Columbia, April 2005. SI3D '05. ACM Press, New York, NY, pp. 95-102.

[13] S. Jeschke, M. Wimmer, H. Schumann, and W. Purgathofer. "Automatic impostor placement for guaranteed frame rates and low memory requirements." *In Proceedings of the 2005 Symposium on interactive 3D Graphics and Games*, Washington, District of Columbia, April 2005. SI3D '05. ACM Press, New York, NY, pp. 103-110.

[14] G. Schaufler and W. Strzlinger, "A three-dimensional image cache for virtual reality." *In Proc. Eurographics*, pp. 227-236, 1996.

[15] Tomas Akenine-Moller, Eric Haines, "Real-Time Rendering", 2nd edition, *A K Peters Ltd*, 2002.

[16] E. H. Adelson, and J. R. Bergen. "The plenoptic function and the elements of early vision." *Computational Models of Visual Processing*, Edited by Michael Landy and J. Anthony Movshon. ISBN 0-262-12155-7. The MIT Press, Cambridge, Mass. 1991, Chapter 1, pp. 3-20.

[17] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. "The lumigraph." *In Computer Graphics Proceedings*, Annual Conference Series, pages 43-54, Proc. SIGGRAPH96 (New Orleans), August 1996. ACM SIGGRAPH.

[18] M. Levoy and P. Hanrahan. "Light field rendering." *In Computer Graphics Proceedings*, Annual Conference Series, pages 31-42, Proc. SIGGRAPH96 (New Orleans), August 1996. ACM SIGGRAPH.

[19] Carnegie Mellon, Mobile camera array, 2004. http://amp.ece.cmu.edu/projects/mobilecamarray/.

[20] Stanford multi-camera array, 2004. http://graphics.stanford.edu/projects/array/.

[21] S. E. Chen, "QuickTime VR An Image-Based Approach to Virtual Environment Navigation." *Computer Graphics (SIGGRAPH95)*, pp. 29-38, August 1995.

[22] J. Foote, and D. Kimber. "FlyCam: practical panoramic video." *In Proceedings of the Eighth ACM international Conference on Multimedia* , Marina del Rey, California, United States. MULTIMEDIA '00. pp. 487-488, 2000.

[23] U. Neumann, T. Pintaric, and A. Rizzo. "Immersive panoramic video." *In Proceedings of the Eighth ACM international Conference on Multimedia* , Marina del Rey, California, United States. MULTIMEDIA '00. pp. 493-494, 2000.

[24] R. Szeliski and H.-Y. Shum. "Creating full view panoramic image mosaics and environment maps." *In SIGGRAPH*, pages 251-258. ACM Press/Addison- Wesley Publishing Co., 1997.

[25] Wikipedia, Panorama definition. http://en.wikipedia.org/wiki/Panoramic_format/.

[26] M. Fiala and G. Roth. "Automatic alignment and graph map building of panoramas". *In IEEE Int. Workshop on Haptic Audio Visual Environments and their Applications*, pp 103-108, October 2005.

[27] VRML - Virtual Reality Modeling Language - Web3D Consortium. 2006 http://www.web3d.org/.

[28] H.Y. Shum and S.B. Kang. "A Review of Image-based Rendering Techniques." *IEEE/SPIE Visual Communications and Image Processing (VCIP) 2000*, pp. 2-13, Perth, June 2000.

[29] S. Chen and L. Williams. "View interpolation for image synthesis." *computer Graphics(SIGGTAPH'93)*, pp. 279-288, august 1993.

[30] S. M. Seitz and C. M. Duer. "View morphing." *In Computer Graphics Proceedings, Annual Conference Series*, pp. 21-30, Proc.SIGGRAPH'96(New Orleans), August 1996. ACM SIGGRAPH.

[31] L. Mcmillan. "An image-based approach to three-dimentional computer graphics." Technical report, Ph.D. Dissertation, UNC Computer Science TR97-013, 1999.

[32] Szymon Rusinkiewicz and Marc Levoy. "QSplat: A Multiresolution Point Rendering system for Large Meshes." *In SIGGRAPH 2000 Conference Proceedings*, pp. 343-352, July 2000.

[33] Lee Westover. "SPLATTING: A parallel, Feed Forward Volume Rendering Algorithm." Ph.D. Dissertation. Technical Report 91-029, University of North Caroline at Chapel Hill. 1991.

[34] J. Shade, S. Gortler, L. W. He, and R.Szeliski. "Layered depth images." *In Computer Graphics (SIGGRAPH'98) Proceedings*, pp. 231-242, Orlando, July 1998. ACM SIGGRAPH.

[35] C. Chang, G. Bishop, and A. Lastra. "LDI tree: A hierarchical representation for image-based rendering." *Computer Graphics (SIGGRAPH'99)*, pp. 291-298, August 1999.

[36] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. "Virtual Network Computing". *IEEE Internet Computing*, 2(1):33-38, January/February 1998.

[37] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. "Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693-702, July 2002.

[38] M. Brachtl, J. Slajs, and P. Slavk, "PDA based navigation system for a 3d environment." *Computers and Graphics*, vol. 25, no. 4, pp. 627-634, August 2001.

[39] I. Yoon and U. Neumann, "Web-based remote rendering with IBRAC (image-based rendering acceleration and compression)." *Computer Graphics Forum*, vol. 19, no. 3, pp. 321-330. 2000.

[40] J. Diepstraten and T. Ertl, "Remote Line Rendering for Mobile Devices." *In Proceedings of Computer Graphics International 2004. IEEE*, 2004, pp. 454-461.

[41] K. Engel, O. Sommer, and T. Ertl. "A Framework for Interactive Hardware Accelerated Remote 3D-Visualization." *In Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym 00*, pages 167-177, May 2000.

[42] Open Inventor Toolkit. http://oss.sgi.com/projects/inventor/

[43] 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.

[44] 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.

[45] 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 (VRAIS96)*, pages 204-213, 1996.

[46] 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.

[47] D. Bradley, A. Brunton, M. Fiala, and G. Roth, "Image-based navigation in real environments using panoramas." *In Proc. of the IEEE Intl Workshop on Haptic Audio Visual Environments and their Applications - HAVE'05*, 2005.

[48] T. Endo, A. Katayama, H. Tamura, M. Hirose, T. Tanikawa, and M. Saito. "Image-based walk-through system for large-scale scenes." *Proceeding of the VSMM'98*, Vol. 1, pp. 269 - 274, 1998.

[49] Y. Lei, Z. Jiang, D. Chen, and H. Bao. "Image-based walkthrough over internet on mobile devices." *In Proc. of Grid and Cooperative Computing - GCC 2004*, pages 728 - 735, 2004.

[50] G. Thomas, G. Point, K. Bouatouch. "A client-server approach to image-based rendering on mobile terminals". *Technical Report*, ISSN 0249-6399, France, January 2005.

[51] L. Yang, and R. Crawfis. "Rail-track viewer: an image-based virtual walkthrough system." *In Proc. of the Workshop on Virtual Environments 2002.* W. Sturzlinger and S. Muller, Eds. ACM Intl Conf. Proc. Series, vol. 23, pp. 37-46, 2002.

[52] A. Boukerche, R. W. Pazzi, J. Feng. "An end-to-end virtual environment streaming technique for thin mobile devices over heterogeneous networks". *Computer Communications*, Volume 31, Issue 11, pp. 2716-2725, 2008.

[53] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. "Protected interactive 3D graphics via remote rendering." *ACM Trans. Graph. 23*, 3, pp. 695-703, (Aug. 2004).

[54] H. Biermann, A. Hertzmann, J. Meyer, K. Perlin, "Stateless Remote Environment Navigation with View Compression", *NYU Technical Report* 1999-784. April 22, 1999.

[55] C. Chang, and S. Ger. "Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering." *In Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia information Processing.* Y. Chen, L. Chang, and C. Hsu, Eds. Lecture Notes In Computer Science, vol. 2532, pages 1105-1111. Springer-Verlag, London, December 16 - 18, 2002.

[56] A. Boukerche and R. W. Pazzi. "Scheduling and Buffering Meshanisms for Remote Rendering Streaming in Virtual Walkthrough Class of Applications". *In Proceedings of the 2nd ACM international Workshop on Wireless Multimedia Networking and Performance Modeling - WMuNeP'06*, pp. 53-60. Terromolinos, Spain, October 2006.

[57] A. Boukerche and R. W. Pazzi. "Remote Rendering and Streaming of Progressive Panoramas for Mobile Devices". *In Proceedings of the 14th Annual ACM international Conference on Multimedia* (Santa Barbara, CA, USA, October 23 - 27, 2006). MULTIMEDIA '06. ACM, New York, NY, pp. 691-694.

[58] Microsoft Corporation, Windows Mobile, http://www.microsoft.com/windowsmobile/en-us/default.mspx.

[59] Microsoft Corporation, C#, http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx.

[60] Microsoft Corporation, DirectX, http://msdn.microsoft.com/en-us/directx/default.aspx.

[61] Microsoft Corporation, Windows Mobile DirectX, http://msdn.microsoft.com/en-us/library/ms172504.aspx.

[62] Microsoft Corporation, .NET compact framework, http://msdn.microsoft.com/en-us/library/f44bbwa1.aspx.

[63] "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms". *IETF RFC* 2001.

[64] H. Shulzrinne, S. Casner, R.Frederick, and V. Jackson. "RTP: A transport protocol for real-time applications". *IETF RFC1889* 1996.

[65] S. Floyd and K. Fall. "Promoting the use of end-to-end congestion control in the Internet". *IEEE/ACM Transactions on Networking, vol. 7(4), pp. 458-472*, august 1999.

[66] S. Floyd, M. Handley, J. Padhye, J.Widmer. "Equation-Based Congestion Control for Unicast Applications". *Proceedings of ACM SIGCOMM 2000*, August 2000, pp 43-56.

[67] S. Biaz and Nithin H. Vaidya. "Discriminating Congestion losses from wireless losses using inter-arrival times at the receiver". *Proceedings of the 1999 IEEE Symposium on Application - Specific Systems abd Software Engineering and Technology.* pp.1017, March 1999.

[68] Song Cen, Pamela C. Cosman, and Geoffrey MVoelker. "End-to-end differentiation of congestion and wireless losses". *IEEE/ACM Transactions on Networking (TON).* 11(5): 703-717, October 2003.

[69] Park, M.K., Sihn, Jun Ho Jeong. "A Statistical method of packet loss type discrimination in wired-wireless networks". *consumer Communications and Networking Conference.* CCNC 2006. 3rd IEEE Volume 1, Issue, 8-10 Jan. 2006 pp. 458-462.

[70] H. Hsiao, A. Chindapol, J.A. Ritcey, Yaw-Chung Chen, Jenq-Neng Hwang. "A new multimedia packet loss classification algorithm for congestion control over wired/wireless channels". *Acoustics, Speech, and Signal Processing, 2005. Proceedings (ICASSP apos;05).* IEEE International Conference on Volume 2, Issue, 18-23 March 2006 pp: ii/1105 - ii/1108 Vol. 2.

[71] OMG: Object Management Group, CORBA , http://www.corba.org.

[72] OpenGL, http://www.opengl.org.

[73] J. El-Sana and A.Varshney. "Generalized view-dependent simplification". textitIn Proceedings of EURO GRAPHICS '99, pp. 83-94, 1999.

[74] J. Lluch, R. Gaitn, E. Camahort, and R. Viv. 2005. "Interactive three-dimensional rendering on mobile computer devices". *In Proceedings of the 2005 ACM SIGCHI international Conference on Advances in Computer Entertainment Technology* (Valencia, Spain, June 15 - 17, 2005). pp. 254-257.

[75] Khronos Group, http://www.khronos.org/opengles/. "OpenGl ES- The standard for Embedded Accelerated 3D Graphics". 2004.

[76] J. El-Sana and Y. J. Chiang. "External Memory View-Dependent Simplification". *In Proceedings of EURO GRAPHICS '00*, pp. 139-150, 2000.x

[77] J. Diepstraten, M. Gorke, and T. Ertl. 2004. "Remote Line Rendering for Mobile Devices". *In Proceedings of the Computer Graphics international (June 16 - 19, 2004).* pp. 454-461.

[78] F. Lamberti, C. Zunino, A. Sanna, A. Fiume, and M. Maniezzo. "An accelerated remote graphics architecture for PDAS". *In Proceedings of the Eighth international Conference on 3D Web Technology* (Saint Malo, France, March 09 - 12, 2003). 55-ff.

[79] T. Jehaes, P. Quax, and W. Lamotte. "Adapting a large scale networked virtual environment for display on a PDA". *In Proceedings of the 2005 ACM SIGCHI international Conference on Advances in Computer Entertainment Technology* (Valencia, Spain, June 15 - 17, 2005). ACE '05, vol. 265, pp. 217-220.

[80] Z. Jiang, Y. Mao, Q. Jia, N. Jiang, J. Tao, X. Fang and H. Bao. "PanoWalk: A Remote Image-Based Rendering Sytem for Mobile Devices". *Proc. Pacific-Rim Conference on Multimedia 2006 (PCM2006)*, Lecture Notes in Computer Science, vol. 4261, pp. 641-649, 2006.

[81] 3D Studio MAX, http://www.autodesk.com.