STREAMING OF HIGH-RESOLUTION PROGRESSIVE MESHES OVER THE INTERNET

WEI CHENG
M.Eng, Fudan University

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERISTY OF SINGAPORE

JANUARY 2010

Acknowledge

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Wei Tsang Ooi, for his continuous support and guidance. Wei Tsang always encourages me to try the new ideas and guides me to the promising direction. He has taught me not only a lot of valuable knowledge but also many skills essential to a researcher, such as writing papers, preparing slides, and making presentations. I am also very grateful to him for his patience in improving my papers, presentations, and thesis. I could not imagine finishing the thesis and the Ph.D. candidature without his help.

I also would like to thank Dr. Kok-Lim Low, who gave me valuable help in computer graphics. I am thankful to Prof. Yong Chiang Tay for his wonderful course and precious advices on system modeling. I would like to thank my thesis committee members, Prof. Tiow Seng Tan and Dr. Kok-Lim Low, and the anonymous external examiner for their constructive criticism to improve my thesis.

I would like to express my gratitude to Dr. Romulus Grigoras and Dr. Géraldine Morin for their continuous help during the collaboration. I would like especially thank Sebastien Mondet. Working together with him is one of the most productive periods in my Ph.D. candidature.

I am also grateful to Ransi De Silva for conducting the user trace collecting experiments and Dan Liu for her help in analyzing the user traces. I appreciate the generous help from Bingjun Zhang in my experiment. I would like to thank all my colleges and friends, including Ma Lin, Nguyen Vu Thanh, Satish Verma, Pavel Korshunov, Balaji Raman, and Ke Liang for their sharing and accompanying.

I can never express enough gratitude to my parents, who love me and support me unconditionally. Without their support and encouragement, I would have not begun the challenging Ph.D student life at my thirties. I also express my thanks to my brother and his family for their supports.

Finally, this thesis is dedicated to my wife, Jingyan Yuan, for her endless caring, understanding, encouragement, and love. I am so fortunate to have her accompanying, without which I could not have finally finished the candidature.

Contents

1 Introduction **1**

1.1 Research Objectives and Scope . . . 4

1.1.1 Quantitatively Determining the Effect of Dependency . . . 4

1.1.2 Receiver-Driven View-Dependent Streaming System of Progressive Meshes 5

1.1.3 P2P View-Dependent Progressive Mesh Streaming System . . . 6

1.1.4 Scope . . . 7

1.2 Contributions . . . 7

1.2.1 Analytical Model of Progressive Mesh Streaming . . . 7

1.2.2 Receiver-driven View-dependent Mesh Streaming System . . . 8

1.2.3 Peer Assisted View-Dependent Progressive Mesh Streaming . . . 9

2 Background and Related Work **11**

2.1 Representation . . . 11

2.2 Coding and Compression . . . 13

2.2.1 Single-rate Coding . . . 13

2.2.2 Progressive Coding . . . 15

2.3 3D streaming . . . 18

2.3.1 Error Resilient Streaming . . . 18

2.3.2 Packetization . . . 19

2.3.3 View-Dependent Streaming . . . 20

3 Analytical Model **22**

3.1 Introduction . . . 22

3.2 Analytical Model . . . 24

3.2.1 Sending Time and Receiving Time . . . 26
3.2.2 Decoding Time of a Vertex Split . . . 27
3.3 Effects of Dependencies . . . 28
3.3.1 Ideal Case . . . 29
3.3.2 Worst Case . . . 29
3.3.3 Insights . . . 34
3.4 Improving Mesh Quality . . . 37
3.4.1 Expected Decoded Mesh Quality . . . 37
3.4.2 A Packetization Algorithm . . . 38
3.5 Model Validation . . . 41
3.5.1 Experimental Setup . . . 42
3.5.2 Validation of Receiving Time and Decoding Time . . . 43
3.5.3 Validation of Quality Curve . . . 48
3.5.4 Validation of the Greedy Method . . . 51
3.6 Conclusion . . . 52

4 Receiver-Driven View-Dependent Streaming of Progressive Meshes 54

4.1 Introduction . . . 54
4.2 Problem Statement . . . 56
4.2.1 Determine Visibility and Visual Contribution of a vertex split . . . 56
4.2.2 Identify Vertices . . . 58
4.2.3 Compress Vertex Splits Efficiently . . . 59
4.3 Receiver-Driven Protocol . . . 60
4.3.1 Mesh Transmission . . . 60
4.3.2 Determining Visual Importance . . . 61
4.3.3 Encoding of Vertex Splits and IDs . . . 64
4.4 Validation . . . 68
4.5 Conclusion . . . 78

5 Characterizing User Interaction with Progressively Transmitted Meshes 81

5.1 Introduction . . . 81

5.2 Experiment Setup 83

5.3 Predictability and Locality 87

5.3.1 Predictability 87

5.3.2 Guiding Caching with User Traces 93

5.4 Generating Synthetic Traces 97

5.4.1 Session Length 97

5.4.2 Think Time 100

5.4.3 Next Action 101

5.4.4 Simplified Model 102

5.4.5 Comparison 106

5.5 Conclusion 108

6 Peer-Assisted View-Dependent Streaming 112

6.1 Introduction 112

6.2 P2P Video Streaming 114

6.3 P2P Mesh Streaming 115

6.4 Hierarchical Chunk Structure 117

6.5 Content Discovery 120

6.5.1 P2P With Centralized Lookup 121

6.5.2 Hierarchical P2P Lookup 123

6.6 Experimental Results 128

6.6.1 Experiment Setup 128

6.6.2 Server Overhead 130

6.6.3 Control Overhead 133

6.6.4 Response Time 133

6.6.5 Summary 134

6.7 Conclusion 134

7 Conclusion 135

7.1 Contributions and Extensions 135

7.1.1 An Analytical Model to Quantify the Effect of Dependency and Packet Loss . . . 135
7.1.2 Receiver-Driven View-Dependent Mesh Streaming Protocol . . . 137
7.1.3 P2P View-Dependent Progressive Mesh Streaming . . . 139
7.2 Other Future Work . . . 140
7.2.1 Coding . . . 140
7.2.2 Transmission . . . 142
7.2.3 Rendering . . . 142

A Related Publications 144

Abstract

High-resolution 3D meshes are increasingly available in networked applications, such as digital museum, online game, and virtual reality. The amount of data constituting a high-resolution 3D mesh can be huge, leading to long downloading time. To reduce the waiting time of users, a common technique for remote viewing is progressive streaming, which allows a low-resolution version of the mesh to be transmitted and rendered with low latency. The quality of the transmitted mesh is incrementally improved by continuously transmitting the refinement information.

Progressive mesh is commonly used to support progressive streaming. A progressive mesh comprises a *base mesh* and a series of refinements. The base mesh is obtained by simplifying the original mesh with a series of *edge collapses*. The inverses of these edge collapses, named *vertex splits*, can reconstruct the original mesh from the base mesh. Therefore, progressive streaming can be implemented by sending the vertex splits as refinements after sending the base mesh.

Streaming of high-resolution progressive meshes is considerably different to video streaming, which has been extensively studied. Frames of a video are usually sent following the time order. Vertex splits of a progressive mesh, however, can be sent in various orders. New research problems arise due to the flexibility in sending order of vertex splits. In this thesis, three such problems are addressed.

First, the progressive coding of meshes introduces dependencies among the vertex splits, and the descendants cannot be decoded before their ancestors are all decoded. Therefore, when a progressive mesh is transmitted over a lossy network, a packet loss will delay the decoding of any following vertex split that depends on a vertex split in this lost packet. Hence, the effect of dependency needs to be considered in choosing the sending order of vertex splits. In this thesis, an analytical model is proposed to quantitatively analyze the effects of dependency by modeling the distribution of decoding time of each vertex split as a function of mesh properties and network parameters. Consequently, different sending orders can be efficiently evaluated without simulations, and this model can help in developing a sending strategy to improve the quality curve during transmission.

Furthermore, this model is applied to study two extreme cases of dependency in progressive meshes. The main insight we found is that if each lost packet is immediately retransmitted upon the packet loss is detected, the effect of dependencies on decoded mesh quality diminishes with time. Therefore, the effect of dependency is only significant in the applications requiring high interactivity.

The accuracy of the analytical model proposed in this thesis is validated under a variety of network conditions, including bursty losses, fluctuating RTT, and varying sending rate. The values predicted from our model match the measured value reasonably well in all cases except when losses are too bursty.

Second, to improve the quality of rendered image in the receiver side quickly, the viewpoint of the user can be considered in deciding the sending order. Non-visible vertex splits do not need to be sent, and the visual contributions of visible vertex splits (how much they can improve the rendered image) also depend on the viewpoint of the user. Hence, choosing the sending order according to the viewpoint of the user, so called *view-dependent streaming*, is a natural choice.

In existing solutions to view-dependent streaming, the sender decides the sending order. The sender needs to maintain the rendering state of each receiver to avoid sending duplicate data. Due to the state-ful design, the sender-driven approach cannot be easily extended to support many receivers with caching proxy and peer-to-peer (P2P) system, two common solutions to scalability.

In this thesis, a receiver-driven protocol is proposed to improve the scalability. In this protocol, the receiver decides the sending order and explicitly requests the vertex splits, while the sender simply sends the data requested. The sending order is computed at the receiver by estimating the visibility and visual contributions of the refinements, even before receiving them, with the help of GPU.

Because the visibility determination and state maintenance are all done by the receivers, the sender in our receiver-driven protocol is stateless and free of complex computation. Furthermore, caching proxy and P2P streaming systems can be applied to improve the scalability without adding more servers.

Third, based on the receiver-driven protocol we proposed, P2P techniques are applied to view-dependent progressive mesh streaming in this thesis. In the implementation of P2P mesh streaming system, two issues are considered: how to partition a progressive mesh into chunks and how to lookup the provider of a chunk. For the latter issue, we investigated two solutions, which trade off server overhead and response time. The first uses a simple centralized lookup service, while the second organizes peers into groups according to the hierarchical structure of the progressive mesh to take advantage of access pattern. We have implemented a prototype and test its performance with synthetic traces we generated based on real traces logged from 37 users. Simulation results show that our proposed systems are robust under high churn rate. It reduces the server overhead by more than 90%, keeps control overhead below 10%, and achieves average response time lower than 1 second.

List of Figures

1.1 Stanford's Digital Michelangelo Project. 1

1.2 Complex digital sculptures created with ZBrush. 2

1.3 Edge collapse and vertex split 3

1.4 Progressive 3D mesh streaming. 4

2.1 Manifold mesh, non-manifold mesh and non-mesh 12

2.2 (a) The triangle strip, (b) the triangle fan, and (c) the generated triangle strip. . 14

2.3 Edge collapse and vertex split 16

3.1 Dependency between vertex splits represented as a DAG. 24

3.2 If packet 0 is lost in the first transmission, compared with the transmissions begin from time slot 1, only the sending order of first T_d packets is different. 30

3.3 The number of decodable packets for ideal case and worst case. 36

3.4 Intermediate quality of decoded mesh. 40

3.5 Loss rate, RTT, and throughput of our traces 43

3.6 Comparing the values of receiving time $E[R_v]$ and decoding time $E[D_v]$ as predicted by our model and as measured from our experiments, using the *Happy Buddha* mesh averaged over 10 runs. 45

3.7 Comparing the values of receiving time $E[R_i]$ and decoding time $E[D_i]$ as predicted by our model and as measured from our experiments, using various meshes. 46

3.8 Comparing the values of receiving time $E[R_i]$ and decoding time $E[D_i]$ as predicted by our model and as measured from our experiments, using various meshes and number of runs. 47

3.9 Comparing the quality curve as predicted by our model and as measured from our experiments, using three different traces. 49
3.10 Comparing the quality curve as predicted by our model and as measured from our experiments, using *Horse* and *Thai Statue*. 50
3.11 Predicted quality curve and measured quality curve from our experiments. 50
3.12 Predicted quality curve and measured quality curve from our experiments. 51
3.13 The comparison of quality curves for FIFO and greedy. 52
3.14 The rendered *Happy Buddha* at the time T_d, using UDP-High. 52

4.1 Sender-driven protocol and receiver-driven protocol 55
4.2 Vertex hierarchy and vertex front. 57
4.3 A vertex split split V_s to V_t and V_u, with V_l and V_r as two cut neighbors. 58
4.4 The process of the sender in receiver-driven protocol. 61
4.5 Screen area 61
4.6 The code of ID of the bottom two vertices is 1011001000. 66
4.7 Meshes we used in the experiments. 69
4.8 Two more viewpoints of Buddha mesh we used. 70
4.9 Two more initial status of Buddha mesh we used. 71
4.10 The quality curve is improved when children vertices are considered when selecting vertices to split. 73
4.11 Effect of updating period on quality curve. 74
4.12 Comparing Screen-area method (weight = 1, 3) with level-based method and random method. 75
4.13 Comparing Screen-area method (weight = 1, 3) with level-based method. Happy Buddha with 60000 vertices are split. 77
4.14 Three parts of the Buddha: head, base, and edge. 78
4.15 The enlarged images of the three parts of the Buddha Mesh 79
4.16 Comparing Screen-area method (weight = 1, 3) with level-based method. Thai Statue with 60000 vertices are split. 80

5.1 Meshes used in experiment. Thai Statue (5 million vertices), Dragon (3.6 million vertices), and Happy Buddha (0.5 million vertices). 82

5.2 The user interface we used to mimic an online shop. 84
5.3 The positive direction of translation and rotation. 86
5.4 An example of a user session 87
5.5 The frequency of 12 actions in the traces we collected. 88
5.6 The conditional probability of next action if the previous action is given. 89
5.7 The probability of repeating the previous action for 9 meshes. 90
5.8 The probability of repeating the previous action for each combination of previous two actions (Thai Mesh). 91
5.9 The number of samples on all viewpoints, and the viewpoints are sorted from the most popular to the least popular. 92
5.10 The number of samples on all combinations of the viewpoint and previous action, and the combinations are sorted from the most popular to the least popular. . . 94
5.11 The accuracy of four prediction methods. 95
5.12 Cumulative Access Count versus Rank 96
5.13 The normalized number of views for each face. 98
5.14 The hit rate when we save part of the faces in the graphic card. 99
5.15 Distribution of Session Length 99
5.16 Think Time (x-axis in log scale). 100
5.17 Think time is smaller when user repeat the previous action. 101
5.18 Two equivalent definitions of state. 102
5.19 The sample size of each combination (previous action, corresponding axis), sorted from the most popular to least popular. Mesh: Thai 104
5.20 Comparison of occurrence frequency of each action. 107
5.21 Comparison of unconditional repeat probability and repeat probability under different previous actions. 108
5.22 Comparison of distribution of viewpoints on different axes. 109
5.23 Comparison of transition probabilities on certain states. 110

6.1 Difference between P2P video streaming, P2P file sharing, and P2P view-dependent mesh streaming. 116
6.2 An example of hierarchical chunk structure. Here $w = 1$, $d = 2$ and $i = 1$. . . . 119

6.3 Centralized Lookup. 122
6.4 (a) Peer B is assigned a leader. (b) An old leader is replaced by peer B. 124
6.5 Peer register in various groups. 126
6.6 The Thai Statue 128
6.7 CDF of the delay and bandwidth used in our simulation 129
6.8 Comparison between centralized lookup approach and hierarchical P2P lookup approach. 131
6.9 Comparison between centralized lookup approach and hierarchical P2P lookup approach. 132

List of Tables

3.1 Notations used in our model. 25
3.2 Main characteristics of the transmission traces. 42
3.3 The characteristics of the different meshes. 44

4.1 The error in visibility determination (before and after force splitting silhouette) . 77

5.1 Introduction of the actions. 86
5.2 The action with highest unconditional probability and its probability. 88
5.3 Session Length and Think Time 99

Chapter 1

Introduction

Advances in 3D scanning technology and 3D modeling techniques have lowered the barrier in creating high-resolution 3D models. High-resolution 3D models can now be created with laser scanner and reconstruction software. For example, the Stanford's Digital Michelangelo Project has digitalized several famous statues of Michelangelo [52] (See Figure 1.1). Similar projects include the Digital Sculpture Project [29] and the project to digitize Rodin's sculptures [58].

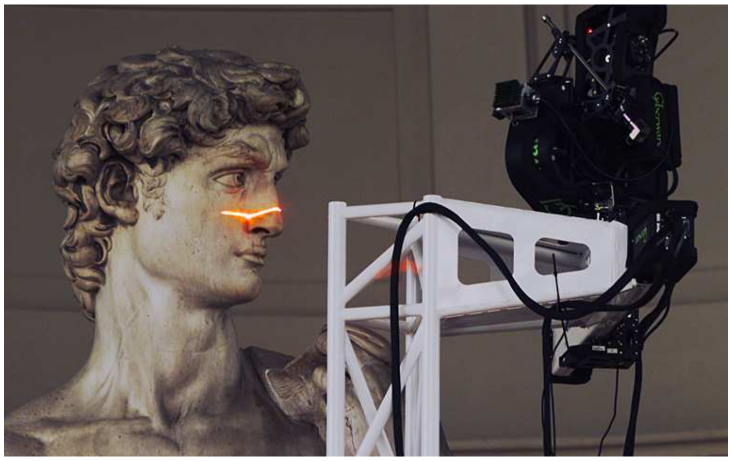


Figure 1.1: Stanford's Digital Michelangelo Project. The head of David Statue is being scanned (image source: http://graphics.stanford.edu/projects/mich).

Further, artists can directly create high-resolution 3D models with digital sculpture softwares such as ZBrush (See Figure 1.2).

Sharing high-resolution 3D models over the Internet is useful in many applications. Digital museum exhibiting 3D models of artifacts is one of the applications. For example, Dr. Andre

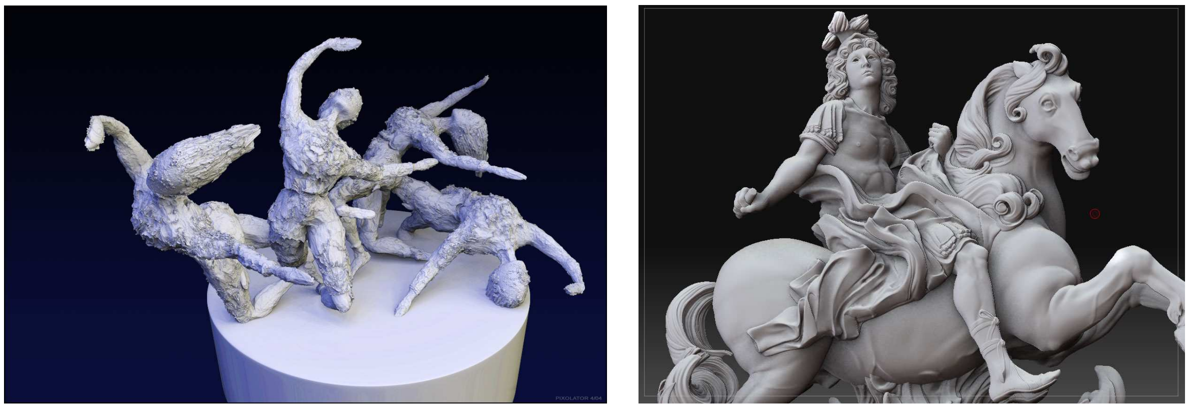


Figure 1.2: Complex digital sculptures created with ZBrush (image source: http://www.evermotion.org/vbulletin/showthread.php?t=64128).

Stork and his teams are working on a project named "History in 3D"[1]. Exhibiting high-resolution 3D models of artifacts instead of the artifacts themselves has many advantages. People can view the artifacts from anywhere, at anytime, and without any travel cost. Visitors can interact with exhibited artifacts in any way they want, without worrying about damaging them or interfering other visitors. Moreover, the museum can exhibit artifacts, however precious they are, safely without worrying about corrosion and theft.

High-resolution 3D models are useful in other networked applications too. For example, virtual world applications, such as Second Life, can become more realistic and interesting by supporting high-resolution 3D models. Commercial users could show their products as 3D models to potential users all over the world. Sculptors could demonstrate their artworks freely without renting a gallery. As another example, virtual earth applications, such as Google Earth, could represent many landmark buildings or statues (e.g. Statue of Liberty) as high-resolution 3D models to give more realistic representation of the real world.

When used in networked applications, high resolution 3D models may take a long time to download completely for display at the client. For example, the Stanford model of the David statue, with 28 million vertices and 56 million triangles, is still 70 MB in size with state-of-the-art compression [3] and needs around 10 minutes to download at 1 Mbps. To reduce the waiting time of the user, streaming techniques can be used to transmit 3D models. The 3D streaming technology enables users to render a coarse version of the model based on partially received data as a preview, and then improve the quality when more data becomes available. Users can

[1]http://www.fraunhofer.de/en/press/research-news/2009/11/history-in-3d.jsp.

decide how much data to receive based on their requirement of quality and the rendering ability of their hardware.

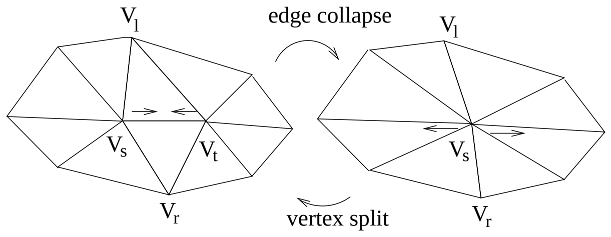


Figure 1.3: Edge collapse and vertex split

One popular representation to support 3D streaming, *progressive mesh*, is proposed by Hoppe [39]. The technique is based on an operation called *edge collapse*, and its reverse operation, *vertex split*. Given a (non-progressive) 3D mesh, the technique applies a series of edge collapses, simplifying the model by reducing the number of vertices and faces. The final, simplified model obtained after this process becomes the *base model*. Given a base model, we can reconstruct the original model by reversing the edge collapse operations through *vertex splits*, incrementally adding new vertices and faces. So, a progressive mesh can be represented by the base model and a series of vertex splits. Figure 1.3 illustrates edge collapse and vertex split operations. Progressive streaming can be implemented by sending the base mesh first as a preview and then sending vertex splits to improve the quality incrementally (See Figure 1.4).

Although audio and video streaming have been studied extensively, 3D streaming remains a new research topic. Video streaming and 3D streaming are significantly different due to the difference between videos and 3D models, and the main difference is how users consume the data. In video streaming, usually the data are consumed in time order. Whereas in 3D streaming, especially when high-resolution 3D models are used, users may consume different subset of the data in various orders.

This flexibility is important in streaming of high-resolution 3D models, because streaming a huge 3D model in a fixed order leads to transmission of data not useful to the user, wasting time and bandwidth. Ensuring this flexibility in streaming systems is an important consideration in choosing representation and coding schemes. The details are discussed in Chapter 2. Meanwhile, this flexibility introduces some new research questions, which we introduce in the next section.

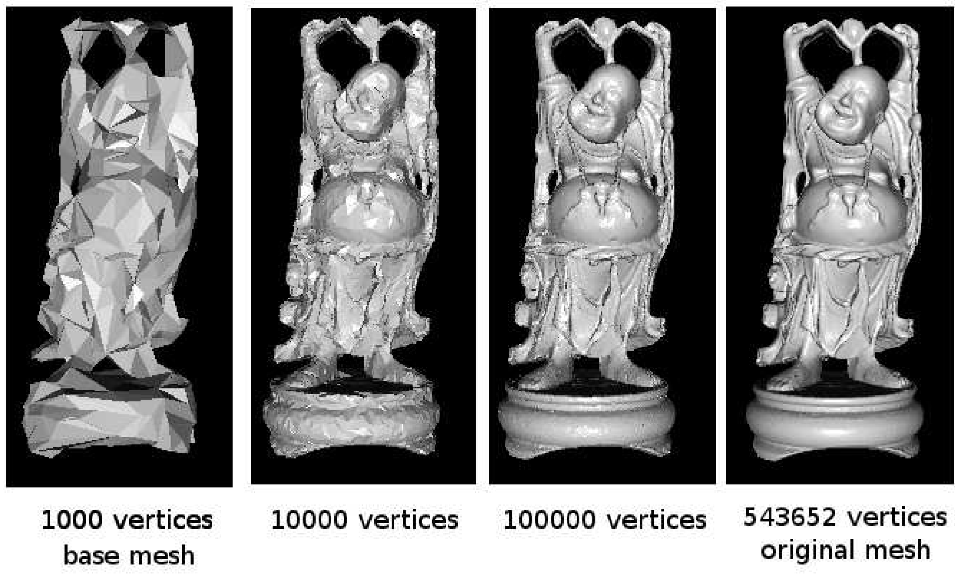


Figure 1.4: Progressive 3D mesh streaming. The original Happy Buddha mesh courtesy of Stanford Computer Graphics Laboratory.

1.1 Research Objectives and Scope

1.1.1 Quantitatively Determining the Effect of Dependency

In progressive streaming, the quality of the mesh on the receiver increases over time. Because vertex splits contribute differently to the quality, how quality improves depends on the decoding order of the vertex splits, which in turn depends on the sending order of the vertex splits. Because of the flexibility in choosing sending order, it is possible to choose a sending order so that the quality on the receiver increases as fast as possible.

A natural method is to send vertex splits in the descending order of their contribution to the quality of the received mesh. This strategy, however, is not always optimal when the mesh is transmitted over a lossy network, because dependency also plays an important role in choosing sending order.

The progressive coding of meshes introduces dependencies among the vertex splits, and the descendants cannot be decoded before their ancestors are all decoded. Therefore, when a progressive mesh is transmitted over a lossy network, a packet loss will delay the decoding of the following vertex splits if they depend on this lost packet. These successfully received vertex splits cannot be decoded until the lost packet is successfully retransmitted. Hence, another consideration in choosing sending order is to minimize the dependency among packets so that most received vertex splits can be decoded without waiting.

Therefore, to choose a proper sending order, it is essential to find a proper trade-off between the importance of vertex and the dependency among packets. Understanding the trade-off requires us to quantify the effect of dependency, which is is non-trivial, as the effect depends on which packets are lost during transmission. Packet losses are random, so the effect of dependency can only be estimated probabilistically.

In summary, the first research objective in this thesis is *to quantitatively analyze the effect of dependency on the quality of reconstructed progressive meshes during transmission over lossy network, and to find a proper sending order to improve the quality as quick as possible, considering both the importance of vertex splits and dependency among the vertex splits.*

1.1.2 Receiver-Driven View-Dependent Streaming System of Progressive Meshes

Another factor, the viewpoint of the user, needs to be considered in deciding the sending order of vertex splits. Sending non-visible data before visible data wastes bandwidth that could be used to send more visible data. Moreover, the visual contributions of visible vertex splits (how much they can improve the rendered image) also depend on the viewpoint of the user. Hence, deciding which vertex splits to send (and in what order) according to the viewpoint of the user, so called *view-dependent streaming*, is crucial to improve the rendered quality of the mesh.

In existing solutions to view-dependent streaming, the sender decides the sending order. This sender-driven protocol has several drawbacks. First, it is not scalable to many receivers as the sender has to determine the visibility of vertices for each receiver. Second, the sender needs to maintain the state of each receiver to avoid sending duplicate data. Due to the stateful design and high computational requirements, the sender-driven approach cannot be easily extended to support many receivers with caching proxy and peer-to-peer system, two common solutions to scalability.

To address the drawbacks of the sender-driven protocol, we want to design a stateless sender for scalable view-dependent progressive mesh streaming, in which the sending order is decided by the receiver. We call this approach the *receiver-driven* approach.

To design a receiver-driven view-dependent mesh streaming system is challenging. The most important challenge is how the receiver determines the visibility and visual contribution of vertex splits before receiving them. The receiver does not have the complete data, unlike the sender. The receiver thus has to estimate the visibility and visual contribution of vertex splits based on partially received data. Moreover, the sender, being stateless, increases the difficulty in compressing data, because now the sender is not aware of the sending order of the vertex splits, and most existing compression algorithms are not applicable since they require specific sending orders. As a result, we need to find an efficient compression algorithm which does not rely on the sending order.

In summary, the second objective in this thesis is *to design and implement a receiver-driven view-dependent streaming system, where the visibility and the sending order are decided by the receiver instead of the sender, thus making the sender stateless and improving the scalability of the system.*

1.1.3 P2P View-Dependent Progressive Mesh Streaming System

The receiver-driven protocol significantly reduces the computing cost at the sender, but the bandwidth can remain the bottleneck if each receiver receives data from the same sender and the number of receivers becomes large. Peer-to-peer (P2P) technique is a common solution to increase the scalability without incurring high infrastructural cost. As a result, applying P2P technique in a view-dependent mesh steaming system enables small companies and individual users to publish their high-resolution 3D models. Hence, P2P technique is essential to popularize applications using high-resolution 3D models.

Applying P2P techniques into progressive mesh streaming system is considerably simplified by our receiver-driven protocol since the sender is stateless and free of expensive computations. The implementation is still challenging due to the flexibility in choosing the sending order and subset to send.

In P2P mesh streaming, users may choose to look at different facets or zoom into different level. Therefore, each peer may choose a unique sending order of vertex splits. Hence, how to group vertex splits into chunks (the data unit for user sharing) is not straightforward. Moreover, since peers may download different subsets of the mesh, it is difficult for a peer to keep receiving needed data from the same peer. A peer would have to continuously look for peers to retrieve the mesh data from, which makes content discovery more challenging.

In summary, the last objective of this thesis is *to investigate chunking and content discovery problems introduced by the flexibility in choosing sending order and sending subset, and to implement a prototype of P2P view-dependent mesh streaming system based on the receiver-driven protocol proposed in this thesis.*

1.1.4 Scope

The scope of our research is as follows.

First, only streaming of static 3D objects is considered in this thesis. Streaming of 3D animations is beyond the scope of this thesis. Second, streaming of 3D scenes is not concerned in this thesis. Instead, this thesis concentrates on streaming of a single large mesh. Nonetheless, streaming of 3D scenes becomes easier when mature methods of streaming of single objects exist because a 3D scene is the collection of multiple single meshes.

This thesis focuses on the streaming of progressive meshes. Streaming of other representations of 3D objects is not discussed in this thesis, but the methods proposed in this thesis should be possibly extended to support streaming of other representations with some modifications.

Textures are not considered in this thesis. Texture coordinates could be assigned to vertex splits similar to the geometry data, so all topics in this thesis could be applicable to progressive meshes with textures (e.g. joint geometry/texture progressive coded meshes [61]) with minor modifications.

1.2 Contributions

In this section, we introduce the contributions of this thesis by summarizing how we achieve the objectives introduced in Section 1.1 and their significances.

1.2.1 Analytical Model of Progressive Mesh Streaming

Our first objective is to quantify the effect of dependency on the rendered quality of progressive meshes when they are transmitted over lossy network, so we can find proper sending order to improve the quality as quick as possible, considering both the mesh property and network condition.

To achieve this objective, an analytical model is proposed to estimate the quality curve based on both the mesh property and network condition. Quality of rendered mesh improves when vertices are split, so we can draw the quality curve when the decoding time of each vertex split is known. From this model, we can efficiently know both the expected value and the distribution of the decoding time of each vertex split before transmission. Consequently, we can analytically evaluate different sending orders without simulations.

Moreover, our model can help in developing a sending strategy to improve the quality as soon as possible, and a greedy strategy is given in the thesis as an example. Further, we derive closed-form expressions in two extreme cases, providing insights to the importance of dependencies on the decoded mesh quality. The main insight is that if each lost packet is immediately retransmitted upon the packet loss detection, then the dependency only matters in the first several round trip times. Therefore, the effect of dependency is only significant in the applications requiring high interactivity.

To show that the insights from the theoretical model can be applied in practice, we extensively verified our model under different realistic conditions, using different progressive meshes, network conditions, and quality metrics. The experimental results show that our model works well under realistic conditions despite the simplification and assumptions we made during modeling.

Our analytical model is general and can be applied to any partially-ordered data without playback deadline. Examples of such partially-ordered data include point-based models organized as QSplat trees [70] and digitized plants. We have actually applied our analytical model to the latter [59].

1.2.2 Receiver-driven View-dependent Mesh Streaming System

Our second objective is to implement a receiver-driven view-dependent streaming system, so that the visibility is decided by the receiver and the sender is stateless.

In implementing the receiver-driven protocol, we need to solve three main problems. First, the receiver has to decide the visual contribution of a vertex split before receiving it. Second, the receiver needs to know the unique ID (identification number) of each vertex so that it can explicitly request data to split it. We avoid explicitly including the IDs in the vertex splits, as it will significantly increase the data size. Third, the vertex splits need to be efficiently compressed without sacrificing the flexibility in choosing sending order.

For the first problem, we find that although it is difficult for the receiver to accurately measure the visual importance of a vertex split before receiving it, estimation suffices in our scheme. We propose a method to estimate the visual importance of a vertex using its screen space area (the area of the neighbor faces of a vertex after projection to the screen). To validate this method, we compared it with two other methods. One is randomly selecting from the visible vertices and the other is based the level of vertex (how many times this vertex has been split). We found that the screen area based method improves the quality significantly quicker than the random method. Moreover, the screen area based method usually generated better rendered images in perception than the level based method.

To uniquely identify each vertex, we borrow the idea from Kim and Lee [47]. In this system, IDs can be deduced by the receiver without any extra information. We also adapt and extend the compression algorithm proposed by Kim et al. [46] to achieve the comparable compression ratio with high flexibility in choosing sending order.

Then, based on these solutions, we implement a receiver-driven protocol in which the visibility determination and state maintenance are all done by the receivers, so the sender in our receiver-driven protocol is stateless and free of complex computation. Hence, caching proxy and P2P streaming systems can be applied to improve the scalability without adding more servers. Therefore, the receiver-driven approach may eventually enable the view-dependent streaming of high-resolution meshes to huge number of receivers.

1.2.3 Peer Assisted View-Dependent Progressive Mesh Streaming

The final objective in this thesis is to implement a prototype of P2P view-dependent mesh streaming system based on the receiver-driven protocol proposed in this thesis.

We compare and contrast P2P mesh streaming to P2P video streaming and P2P file downloading and point out the main difficulties of P2P mesh streaming: chunking and content discovery.

We propose a chunking scheme by exploiting the hierarchical nature of the data. It suits mesh streaming well. First, the receiver can know which vertex split belongs to which chunk implicitly without extra data embedded in the stream. Second, vertex splits in a chunk are close to each other so their visibility are similar. Third, the dependency among chunks are minimized.

We investigate two content discovery schemes for P2P mesh streaming: centralized lookup and hierarchical P2P lookup. The first design is easy to implement and works well in small to middle scale network. The structure of the progressive mesh is considered in the second design to further reduce server overhead, so it is more suitable for large scale networks.

Finally, we run simulations with a large number of synthetic traces, generated based on real traces we collected from 37 users, to evaluate the P2P mesh streaming system we proposed. Analysis and simulation results show that server overhead can be reduced by 90%. Meanwhile, average response time and control overhead are low.

Our results indicate that with the help of P2P technique, sharing high-resolution 3D models to a large number of users over the network is more affordable to small companies and even individuals. We believe this is essential for networked applications using high-resolution 3D models to become popular.

The rest of the thesis is organized as follows. In Chapter 2, we introduce some backgrounds and related works. The subsequent chapters introduce how we achieve our objectives in detail. Chapter 3 introduces our analytical model and its applications. Chapter 4 describes our receiver-driven view-dependent streaming protocol in detail. We include the introduction of how we collected the real user traces and generated the synthetic traces in Chapter 5 for completeness. Then, we introduce the P2P mesh streaming systems we proposed in Chapter 6. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Background and Related Work

2.1 Representation

A 3D object can be represented with different models. The most popular representation of 3D objects is polygonal meshes, especially the triangle meshes. Point-based representations are also used in practice. A point-based representation can be considered as a sampling of a continuous surface, resulting in a set of 3D points. To bridge the gaps between neighboring point samples, bounding spheres [70, 71] or surface splats [92] are used as rendering primitives. More details about point-based models can be seen in the survey by Kobbelt and Botsch [49] and the PhD thesis of Wand [82]. In addition to these surface modeling methods, constructive solid geometry (CSG), a commonly used technique used in solid modeling, constructs visually complex 3D objects by applying boolean operations (such as union, intersection, and difference) on simple solid objects, such as cubes, spheres, cylinders, cones, and pyramids. CSG is extensively used in game development and CAD applications, because it is relatively easy to be constructed manually with computers and provides mathematical accuracy. As a specific example, we have used generalized cylinders to represent and streaming plants [59, 60].

This thesis concentrates on meshes for the following reasons:

- Most rendering hardware are optimized for meshes, so other model needs to be convert to meshes before rendering either explicitly or implicitly.

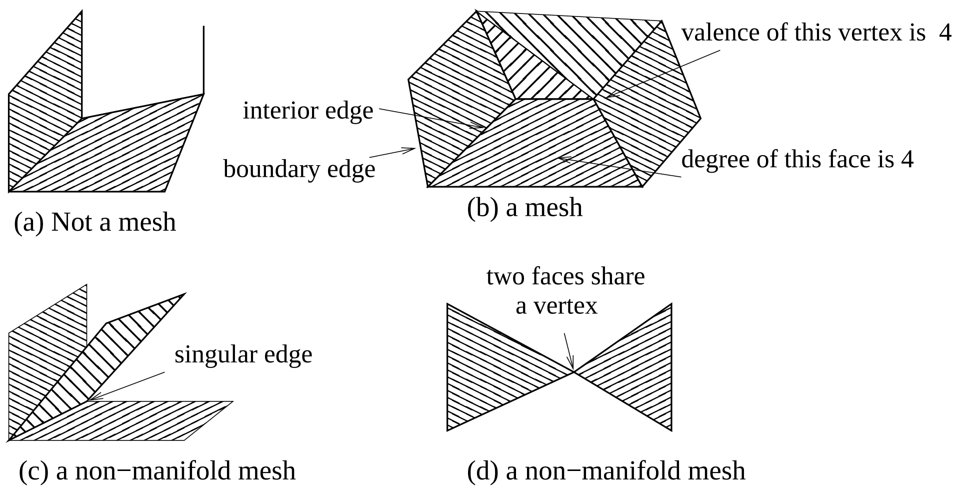


Figure 2.1: (a) is not a mesh since one edge is incident to no face. (b) is a manifold mesh. (c) is a non-manifold mesh since there is a singular edge. (d) is a non-manifold mesh since two faces share only a vertex.

- Without connectivity information in the model, point-based models are relatively easier to deal with than meshes. Therefore, adapting techniques used in streaming meshes to streaming point-based model is easier than the opposite direction.

A triangle mesh can be represented as a tuple (K, V), where K is a *simplicial complex* including all the mesh simplices (vertices, edges and triangles) and $V = \{v_1, v_2, ..., v_n\}$ is the set of geometry positions of vertices. Therefore, a mesh has both *connectivity* information represented by K and *geometry information* represented by V. In other words, K includes all the topology information of all the elements in the mesh. If polygons are allowed in K instead of only triangles, then this kind of meshes are called polygonal meshes.

In a mesh, it is required that every vertex is an end of an edge and every edge should be incident to at least one face (Figure 2.1(a) is not a mesh). The number of edges incident to a vertex is called the *valence* of this vertex, and the number of edges incident to a face is called the *degree* of this face. An edge that is incident only to one face is called a *boundary edge*; an edge shared by two faces is called an *interior edge*; an edge shared by more than two faces is called a *singular edge.* A *manifold mesh* is a mesh without singular edges and every two faces share one edge or nothing (see Figure 2.1).

Polygonal meshes can be used as an approximation of the surfaces of 3D objects. In this case, some properties of the surface such as color and normal are often associated with the mesh.

These properties can be categorized into two types: discrete attributes and scalar attributes. Discrete attributes, such as material identifier, are usually associated with the faces, while scalar attributes, such as color, normal, and texture coordinate, are often associated with vertices or corners (the tuple (vertex, face)). The latter one is used because in the case of discontinuity, a vertex can have more than one scalar value. For example, if two smooth surfaces intersect at a curve, then vertices in this curve will have two different normal values. Therefore, the normal value should be associated with the tuple (vertex face). Then, polygonal meshes can be represented by (K, V, D, S) where D is the set of discrete attributes associated with face $f \in K$ and S is the set of scalar attributes associated with corners (v, f) in S.

2.2 Coding and Compression

A mesh is encoded first before it can be stored or transmitted. Coding represents a mesh with a sequence of bits. Compression is always combined with coding to save the storage space or the bandwidth needed in transmission. There are two kinds of coding: single-rate coding and progressive coding. Single-rate coded mesh can only be decoded when all data becomes available whereas progressive coded mesh can be decoded with only part of the data is available. We will briefly introduce single-rate coding methods, and then concentrate on progressive coding methods, because the latter are more suitable for progressive streaming.

2.2.1 Single-rate Coding

In a mesh, three kinds of information need to be coded: connectivity, geometry and property. Since most property data are associated with vertices, many compression algorithms either treats them as geometry data or ignore them. We first introduce connectivity coding, and then introduce geometry coding briefly.

A triangle mesh is often represented as an array of vertex coordinates (geometry information) and an array of triangles (connectivity information). A triangle is represented by the indices of its three vertices, where every vertex is indexed by all the triangles to which it is incident. Repeated references to the triangles introduce redundancy. Such redundancy can be reduced using either triangle strip, a triangle fan, or a generalized triangle strip (a mixture of triangle strips and triangle fans) (see Figure 2.2), in which triangles are represented with one index of vertex except the first triangle. Deering [30] first introduced generalized triangular mesh, which combining the generalized triangle strips with a vertex buffer to further reduce the bits used to represent indices. Furthermore, Chow [24] introduced a method to decompose a mesh into triangle strips. Bajaj, Pascucci and Zhuang [11] presented a coding method with layered decomposition. Typically, a triangle layer is a generalized triangle strip.

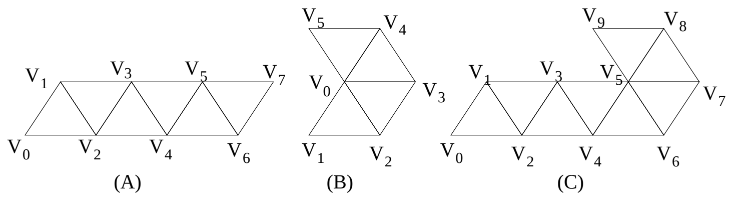


Figure 2.2: (a) The triangle strip, (b) the triangle fan, and (c) the generated triangle strip.

Taubin and Rossignac [77] proposed a method named *topological surgery*, which changes coding of a manifold mesh to coding of two trees: a vertex spanning tree and the dual of the triangles. Topological surgery has been implemented in MPEG-4 standard, and more details can be seen in the report of Taubin [75].

Touma and Gotsman [80] introduced a valence-driven approach to encode the connectivity by coding the valence of all the vertices in a specific traversing order. The performance is further improved by Alliez and Desbrun [5], who also proved a upper-bound 3.24 bpv, exactly the theoretical value computed by Tutte [81] after enumerating all possible planar graphs.

Gumhold and Straßer proposed a method called *cut-border machine* [36]. A significant advantage of this method is that it can be easily implemented in hardware and the decompression is fast, which makes it very useful for real-time coding. Another method named *edgebreaker* is presented by Rossignac [69]. It is similar to the cut-border machine but has better compress efficiency and much slower decompression speed.

After this introduction of connectivity coding, we briefly introduce geometry coding. Most single-rate geometry compression methods involve three steps: quantization, prediction, and entropy coding.

In VRML, geometry data is represented with 32 bit floating-point numbers, but this precision is beyond the perceiving capability of human eyes. Hence, quantization can be performed to compress the geometry data. Typically, each coordinate in a mesh is uniformly quantized to 8-16 bits integer.

After quantization of vertex coordinates, vertex coordinates are predicted by exploiting correlations between adjacent vertex coordinates. An effective prediction scheme generates prediction errors with highly compact distribution, which can be effectively encoded by entropy coders, such as Huffman coder or arithmetic coder.

More details about single-rate coding can be seen in the tutorial by Gotsman, Gumhold, and Kobbelt [33], the survey by Taubin [75], by Alliez and Gotsman [6], and the survey by Peng, Kim and Kuo [66].

2.2.2 Progressive Coding

Single-rate coded meshes can only be decoded after they are completely received, so they are not suitable for streaming. Therefore, some progressive coding schemes have been proposed. According to how the original mesh is simplified, these coding schemes can be categorized into three classes: progressive mesh-based, vertex clustering-based, and re-meshing-based.

Vertex Clustering-Based In vertex clustering based schemes, space is divided into cells following regular structure such as kd-tree [32] and octree [67]. All vertices inside a cell are represented by one delegate. Cells can be further divided into smaller cells and the quality of reconstructed model increases. Vertex clustering based coding can code geometry information efficiently, but it is difficult to code connectivity information. Moreover, the quality of base meshes is poor.

Re-meshing-Based Re-meshing-based methods only keep the shape of a model and does not keep the accurate topology. An irregular mesh can be converted into a semi-regular mesh with similar shape. Because of the regularity of the new mesh, the connectivity information can be coded almost without cost. In 2002, Gu et al. proposed an algorithm to remesh a mesh to a completely regular structure and represent it as a *geometry image* [34]. Geometry, color, and normal information of a 3D mesh then can be stored as 2D images. Traditional image processing techinique can then be used. For example, spectral geometry processing are studied by many researchers (e.g. Taubin [74], Karni and Gotsman [45]). A state-of-art survey was done by Zhang et al. [89]. He et al. proposed *spectral geometry images* to generate more visually pleasing shapes with higher compression efficiency.

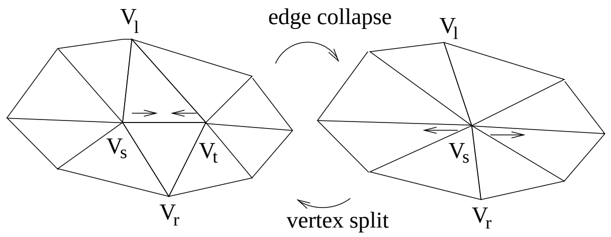


Figure 2.3: Edge collapse and vertex split

Moreover, they proposed an error protection scheme to target networks with packet loss [38].

Progressive Mesh-Based In this thesis, we concentrate on progressive meshes, which provide the finest granularity in choosing streaming order and subset, so we introduce the progressive mesh based coding in more details.

The progressive mesh (PM) is first introduced by Hoppe in 1996 [39]. In this scheme, the simplification is based on *edge collapse*, which collapses an edge into a vertex (see Figure 2.3). After a series of edge collapses, a coarser *base mesh* remains. Then the PM is constructed by the base mesh and a series of *vertex splits*, the inverse of edge collapses, in a reverse order. Different level-of-detail can be achieved by applying different numbers of vertex splits.

The base mesh can be encoded with the single-rate coding methods introduced in section 2.2.1. Here, we introduce how to encode the vertex splits. For connectivity information, we need to code s, l, and r, and for geometry information, we need to code the new position of v_s and the position of v_r. If half collapse, which always collapse an edge to one of its ends (i.e. v_s keeps unchanged), is used, we only need to code the position of v_r.

If the number of vertices in the current model is V, then s can be any one of them, and $\lceil log_2 V \rceil$ bits are needed to encode s.

Many compression methods are proposed to reduce this cost. Hoppe [41] present an efficient implementation, in which the vertex splits are ordered such that the next one is as close to current one as possible. This method reduces the cost from $O(V log_2 V)$ to $O(V)$. Karni, Bogomjakov, and Gotsman [44], also use a specific order of vertex splits to reduce the cost of connectivity. The vertex sequence is generated by recursively cutting a mesh into two balanced parts with minimum cut. With their algorithm, the connectivity can be encoded with in average 4.5 bpv.

Another idea is to group many vertex splits together, and efficiency can be improved at the cost of coarser granularity. Taubin, Guéziec, Horn, and Lazarus [76] proposed the progressive forest split scheme (PFS), in which several edge collapse operations are combined together and the inverse operation splits a forest of edges and vertices. Payarola and Rossignac [63], proposed compressed progressive mesh (CPM) with similar idea. They group about 50% vertex splits into one batch. To differentiate vertices to be split with others, one bit per vertex is used as indicative. Other methods have a similar idea but are based on vertex removal (remove a vertex and triangulate the hole generated) [27].

More methods exist to reduce the cost in encoding connectivity information. Some of them are extended from the single rate coding algorithms [10, 4]. Some extend progressive mesh to directly encode non-manifold meshes [68].

All compression methods introduced above have a common drawbacks. After compression, the data becomes linear and has to be decoded in a fixed order. Therefore, these methods improve the coding efficiency at the cost of losing flexibility, which is essential in streaming of high-resolution progressive meshes.

Even without compression, flexibility of choosing streaming order is still restricted by the dependency among the vertex splits. For a manifold mesh, a vertex split operation depends on the existence of

- the vertex to be split (V_s in Figure 2.3),
- two cut neighbors (V_l, V_r in Figure 2.3).

More dependencies exist if artificial folds are strictly forbidden [40, 56], but in this thesis we ignore these dependencies since we can tolerate temporary folds in our scheme.

To et al. [79] further removed the second dependency. In their method, if a cut neighbor does not exist during a vertex split, its ancestor are used as the cut neighbor instead. Kim and Lee [47] improved this method so that the final mesh can keep the original connectivity. Kim et al. [46] proposed a better scheme that enables an ordinary progressive mesh to be split in random order.

These methods increase the flexibility, but reduce coding efficiency. One solution proposed by Yang et al. [85] is to divide the whole mesh into several segments and encode them separately to trade off between flexibility and compression efficiency. The weakness is that the size of the base mesh is relatively large since the original vertices in the border of segments are kept in the base mesh. Furthermore, the quality of the base mesh is uneven.

Kim et al. [46] have proposed a compression algorithm that allows random splitting of a mesh without sacrificing compression efficiency. This method is neat since it achieves comparable efficiency with other compression methods but maintains the highest flexibility. This algorithm, however, is not designed for network transmission. We extend this algorithm in this thesis. More details are introduced in Chapter 4.

2.3 3D streaming

Given the backgrounds in modeling and coding, now we introduce some related works in 3D streaming. The main concern in 3D streaming is how to improve the quality of the received mesh as fast as possible. Current studies can be categorized into three groups by how to achieve this objective.

2.3.1 Error Resilient Streaming

To improve the quality quickly, we need to mitigate distortion of the rendered quality of a rendered mesh in the presence of packet losses.

At the transport layer, Al-Regib and Altunbasak [2], Li et al. [53, 54], and Chen et al. [17] have investigated how to intelligently select either TCP or UDP for transmissions to trade off reliability and end-to-end delay. Li et al. have also considered SCTP with partial reliability. Harris III and Kravets [37] proposed a new transport protocol that exploits loss tolerance and partially-ordered property of 3D objects organized into trees of bounding volumes.

At the application layer, the major error control techniques: error resilient coding, error protection, retransmission, and error concealment [64], have all been applied to mesh streaming.

Existing work in robust mesh compression aims to reduce dependencies among the mesh [65, 84]. Similar to introducing key frames or restart markers in video/image coding, mesh segmentation is used to reduce the affected range of one packet loss. In robust mesh compression, a mesh is typically partitioned into several independent segments and then coded separately. Therefore the effect of one packet loss is confined to the segment to which it belongs. The finer the partition is, the fewer the affected vertices are. The coding efficiency, however, decreases due to more redundancies and less correlation.

Al-Regib and Altunbasak [1] proposed an unequal error protection method to improve the resilience of progressive 3D mesh based on CPM (compressed progressive meshes). Forward error correction (FEC) codes are added to the base mesh and additional levels-of-detail information to maximize the decoded mesh quality. The method is similar to FEC protection of video data.

Chen et al. [17] also applied FEC to streaming progressive meshes. They analyzed several transmission schemes: TCP only, UDP only, TCP with UDP, and UDP with FEC, and studied their effects experimentally on the transmission time and decoded mesh quality.

Cheng et al. [18] proposed a sending strategy of mesh with textures based on sub-sampling. Their objective is to optimize the perceptional quality, which depends on both geometry quality and texture quality.

Some studies use retransmission to recover the packet loss. For example, Tian [78] uses selective retransmission in their system. This thesis concentrates on retransmission because it can be argued that retransmission is more suitable than FEC when the feedback channel is available, because there is no playback deadline in streaming of 3D meshes and data is always useful.

Most of the methods introduced above treat the meshes streamed as traditional linear data. When the flexibility of progressive mesh is considered, a new question arises: how to select a sending order to improve the quality quickly and mitigate the effect of packet loss. We call it the *packetization* problem.

2.3.2 Packetization

Packetization is tackled by Harris III and Karvets in [37]. They proposed a protocol named On-Demand Graphic Transport Protocol (OGP) for transmitting 3D models represented as a tree of bounding volumes. A key component of the protocol is to decide which bounding volumes to send. OGP begins with packing the largest possible subtree at the root and continues to pack the nodes in the subtree of acknowledged nodes in breadth-first order.

Gu and Ooi [35] were the first to look at the packetization problem for progressive meshes. They model the packetization problem as a graph problem where the objective is to equally partition the graph into k partitions with minimum cut size. The problem is shown to be NP-complete and a heuristic is proposed. They, however, assume that every vertex split contributes equally to the rendered quality.

In practice, the contribution of vertex splits can vary considerably. Therefore, when choosing sending order, we need to trade off between minimizing the dependency and sending vertex split with higher contribution first. To achieve this objective, the effect of dependency needs to be quantitatively measured. No existing study is done for this problem prior to this thesis.

A commonly used metric of contribution of vertex splits is Hausdorff distance between the original and reconstructed mesh [26]. As Hausdorff distance is view independent, bandwidth may be wasted in sending invisible vertex splits before the visible ones. Moreover, even among the visible vertex splits, the view-independent metric cannot reflect the real contribution to the visual quality of clients with different viewpoints. A vertex split that significantly changes a mesh may change the rendered image only slightly. A better metric for visual contribution of a vertex split, based on the rendered image on the screen, needs to consider the receiver's viewpoint. Streaming of 3D models based on this metric is named *view-dependent* streaming.

2.3.3 View-Dependent Streaming

The view-dependent approach first appeared as a dynamic simplification method used for adaptive rendering of a complex 3D mesh [40, 56]. Only vertex splits that contribute to the rendered image will be rendered, allowing real-time rendering of a complex mesh even with limited rendering capability. Besides progressive mesh, other multi-resolution representations, such as vertex-clustering and subdivision scheme, are used in view-dependent refinement systems [83, 7, 9].

Later, the view-dependent approach is used in progressive streaming of 3D meshes. In the scheme proposed by Southern et al. [73], the client is stateless and maintains only the visible data. To et al. [79] and Kim et al. [48] proposed that received data are stored in the receiver even after they become invisible, so they need not be resent when they are visible again. In these papers, view-dependent approaches mainly aim at addressing limited rendering capability.

Yang et al. [85] and Zheng et al. [91], on the other hand, use view-dependent streaming to address limited network bandwidth. Yang et al. proposed a scheme where the server chooses the appropriate resolution according to the available network bandwidth. Zheng et al. [91] predict the user action to request data in advance in order to reduce the effect of network latency and compensate the round-trip delay with the rendering time. Moreover, Yang et al. [86] proposed a joined optimizing transmission method based on a general rate-distortion model that considers both mesh and texture data. Their method allocates bits between mesh and texture to optimize the display quality at every stage of progressive transmission.

These systems use a sender-driven approach and do not address server scalability issues. Due to the stateful design, the sender-driven approach is difficult to be extended to support caching proxy and peer-to-peer techniques, two common solutions to scalability. This thesis presents the first scalable view-dependent streaming system without incurring high infrastructural cost.

Chapter 3

Analytical Model

3.1 Introduction

Progressive coding of meshes, much like progressive coding of video, introduces dependencies between data. Dependencies between data units can cause delay in decoding when sent over a lossy network – a data unit cannot be decoded and displayed if one of the other data units it depends on is not received correctly. For example, in the context of video streaming, MPEG-encoded frames are inter-dependent: an I-frame has to be received and decoded properly before being able to decode the subsequent P-frames and B-frames referencing this I-frame (either directly or indirectly). Another example is in layered coded video, where the base layer has to be received before the enhancement layers can be decoded. The effects of dependencies in the context of video streaming have been well studied in the literature [14]. Multi-resolution 3D objects also have dependencies between different level of details. For progressive meshes, the mesh is refined by successive vertex splits. Thus, dependencies exist between the original vertex and its one ring (the direct neighbors of the vertex) and the vertices and triangles created by the vertex split. The effects of these dependencies on decoded mesh quality, however, are not well understood. Due to the fundamentally different nature of progressive mesh and video data, what we learn from video streaming research does not apply. This chapter aims to study the effect of dependencies in progressive mesh streaming by proposing an analytical model, relating the decoded mesh quality of progressive meshes to packet loss rate, given the dependencies among vertex splits.

The decoded mesh quality at a given time t is determined by the vertex splits that are decoded before t and their contribution to the overall mesh quality. To estimate the decoded mesh quality at time t, our analytical model predicts the decoding time of each vertex split. The total contribution of vertex splits decodable before t gives the quality. To predict the decoding time, we first express the expected time for receiving each packet in terms of loss rate, round trip time, and sending rate. A received vertex split has to wait for the vertex splits it depends on to arrive before it can be decoded. Therefore the received time of a vertex split is not always equal to its decoding time. Our model gives an expression for the expected decoding time given the dependencies among the vertex splits. Furthermore, we propose a metric to measure the contribution of vertex splits to the decoded mesh quality. The decoded mesh quality then can be computed after we know the expected decoding time of every vertex split and its contribution.

Our analytical model is useful in several ways. First, we can analytically evaluate different strategies for streaming a progressive mesh. For instance, packetization of vertex splits affects the intermediate decoded mesh quality at the receiver. Evaluating different packetization scheme using simulation is an option, but it may take many experiments to obtain accurate expected values. Our model computes the expected value easily. Second, our model can also help in developing a better sending strategy. The quality of the decoded mesh depends on various factors, which include not only network conditions such as loss rate and round trip time, but also the order, the dependency, and the importance of the data. The last three factors are in turn affected by the packetization strategy. Our model can estimate the effect of each factor on the decoded mesh quality. As such, we can make the right trade-off between these factors during transmission to improve the quality. Finally, we derive closed-form expressions in two extreme cases, giving us insights into the importance of dependencies on the decoded mesh quality.

Our analytical model is general and can be applied to any partially-ordered data without playback deadline. We choose to concentrate on progressive 3D meshes (which are indeed partially ordered chunks) in this chapter. While meshes are not necessarily the best 3D representations, they are popular and support finer granularity of progressivity than other multi-resolution representations. Other examples of such partially-ordered data include point-based models organized as QSplat trees [70] and digitized plants. We have actually applied our analytical model to the latter [59].

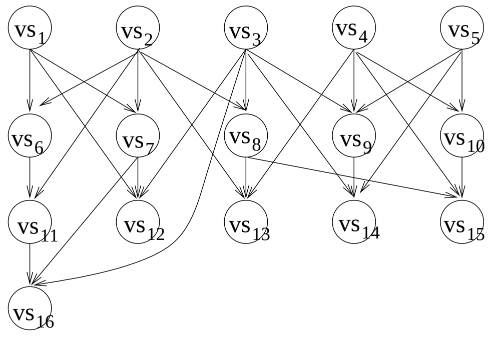


Figure 3.1: Dependency between vertex splits represented as a DAG.

The rest of this chapter consists of five sections. Section 3.2 introduces our analytical model. We then describe the applications of our model. In Section 3.3, we study two hypothetical extreme cases, and show the effect of dependencies on decoded mesh quality. In Section 3.4, we show how our model can be used to analytically compute the expected decoded mesh quality and how this expected quality can be used to design a packetization strategy for transmitting progressive meshs. Section 3.5 presents validation of our model and evaluation of the proposed packetization strategy, using network traces using UDP and DCCP under various network conditions. Section 3.6 concludes by reflecting on the insights we gain from our model and its implications.

3.2 Analytical Model

We now develop an analytical model for transmission of progressive mesh over lossy networks. We first need to model the progressive mesh and the network. Since typically the base model is small (less than 1% of the total size) and is transmitted using reliable transport protocols [2], we assume that the base model has been received by the receiver. We will focus on modeling the vertex splits, which can be represented as a directed acyclic graph (DAG) (see Figure 3.1). In the DAG, nodes represent the vertex split operations, and edges represent the dependency between these operations. We assign each node a weight, which corresponds to the importance of the vertex split. We will use the terms node and vertex split interchangeably in this chapter.

On the sender, vertex splits are grouped into packets, each containing equal number of vertex splits. We say packet P is a *parent packet* of the vertex split v if v depends on at least one vertex split in P. Therefore, a vertex split can be decoded only after the packet containing it and all its parent packets are received.

Let B be the average sending rate[1] and p be the packet loss rate. In the model, we assume that packet losses are independent to each other for simplicity. In Section 3.5, we show that our model still works well under realistic packet losses (more bursty). We assume a retransmission-based protocol – the sender resends a packet immediately when its loss is detected. Let T_d be the average period between the time a packet is sent and the time its loss is detected at the sender (hence the time it is retransmitted). While B, p, and T_d are likely to vary in practice due to network dynamics, we adopt a common technique in performance modeling by constructing our model using their average values and assuming that they are constants. We study the effects of this assumption in Section 3.5.

We discretize time into slots. Each slot is the time to send one packet. Thus, one time unit is L/B seconds, where L is the packet size. The time T_d is normalized to this time unit. Thus, T_d can also be interpreted as the number of packets transmitted between sending a packet P and detecting the loss of P. Moreover, at the sender, we define the time the first packet is sent as time 0, whereas, at the receiver, we define the time the first packet is received (if it is not lost) as 0. Therefore, a packet sent at time t will be received at time t if it is not lost. Thanks to the different timeline at the receiver, we avoid an additional term, the propagation time of a packet sent from the sender to the receiver, in equations related to the received time and decoded time. We denote the i-th packet sent (and not retransmitted) as P_i, where $i = 0, 1, ...$ Table 3.1 summarizes these notations as well as other major notations used in this chapter.

L	packet size
B	sending rate
p	packet loss rate
T_d	time between sending a packet and receiving its NACK
S_i	sending time of P_i (wrt. sender time)
R_i	received time of P_i (wrt. receiver time)
D_j	the time vertex split j is decoded at the receiver
w_j	the importance of vertex split j
$N_{d,t}$	number of packets decoded at time t
$N_{r,t}$	number of packets received at time t
Δ_t	increase in expected number of decodable packet at time t

Table 3.1: Notations used in our model.

[1]decided either by the available bandwidth or by TCP friendly requirement

With that background, we are able to explain how we obtain the expected value of sending time, received time, and decoding time of a vertex split.

3.2.1 Sending Time and Receiving Time

Let S_i be the time when P_i is sent. Lemma 1 computes the distribution of S_i and the expected value $E[S_i]$.

Lemma 1 *If $i < T_d$,*

$$S_i = i.$$

Otherwise,

$$E[S_i] \quad = \quad (i - T_d + 1)\frac{1}{1-p} + T_d - 1.$$

Proof If $i < T_d$, then $S_i = i$ since no retransmission exists. Now we consider the case when $i \geq T_d$. After time T_d, a slot t will be used in retransmission if packet sent in $t - T_d$ is lost. As a result, a new packet can only be sent if a packet sent in $t - T_d$ is successful. If packet i is sent at time S_i, then $i - (T_d - 1)$ new packets are sent after time T_d. In other words, there are $i - T_d + 1$ known successful transmissions. The number of transmissions for a transmission to succeed is a random variable with geometric distribution and its expected value is $1/(1-p)$. Therefore, the expected value of transmissions when there are $i - T_d + 1$ known successful transmissions is $(i - T_d + 1)/(1-p)$. Notice there are also $T_d - 1$ recently sent packets whose status we do not know, we have

$$E[S_i] = (i - T_d + 1)\frac{1}{1-p} + T_d - 1.$$

$\square$

Let R_i denote the time a packet i is received. The probability that a packet i is received at time t is given as follows.

Lemma 2

$$Pr(R_i = t) = \begin{cases} (1-p)p^{n_{i,t}} & \textit{if } (t - S_i) \mod T_d = 0 \\ 0 & \textit{otherwise} \end{cases}$$

where $n_{i,t} = \lfloor (t - S_i)/T_d \rfloor$ is the number of times packet i was lost when $R_i = t$.

Proof The lemma follows from the fact that a packet can only be received at a time that is a multiple of T_d after the first time it was sent. $\square$

Note that here S_i is a random variable but we approximate the decoding time by using the expected value of S_i computed from Lemma 1. This approximation is accurate enough (as shown in the next subsection) as long as the variance of S_i is small.

Lemma 3

$$\begin{aligned} Pr(R_i \le t) &= 1 - p^{n_{i,t}+1}; \\ Pr(R_i < t) &= \begin{cases} 1 - p^{n_{i,t}} & \textit{if } (t - S_i) \mod T_d = 0 \\ 1 - p^{n_{i,t}+1} & \textit{otherwise} \end{cases} \end{aligned}$$

Proof A packet is received strictly after time t if and only if the packet is lost at least $n_{i,t}+1$ times and the probability is $p^{n_{i,t}+1}$. Then, we have $Pr(R_i \le t) = 1 - p^{n_{i,t}+1}$. Considering $Pr(R_i < t) = Pr(R_i \le t) - Pr(R_i = t)$ and Lemma 2, Lemma 3 follows. $\square$

3.2.2 Decoding Time of a Vertex Split

Once we expressed both sending time and receiving time, we can approximate the decoding time of a vertex split v, denoted as D_v.

Let $\mathcal{P}(v)$ be the set of all parent packets of the vertex split v and the packet including v, then the probability that vertex split v can be decoded at time t is given by the probability that one of the packets in $\mathcal{P}(v)$ is received at time t and all other packets in $\mathcal{P}(v)$ are received before time t.

$$Pr(D_v = t) = \sum_{i \in \mathcal{P}(v)} \big(\frac{Pr(R_i = t)}{Pr(R_i < t)} \prod_{k \in \mathcal{P}(v)} Pr(R_k < t) \big). \tag{3.1}$$

Lemma 2 and 3 give the expression for $Pr(R_i = t)$ and $Pr(R_i < t)$ respectively. Once we have the probability distribution of D_v, we can estimate the expected decoding time of a vertex split v with

$$E[D_v] = \sum_{j=t}^{\infty} j Pr(D_v = j). \tag{3.2}$$

Since the probability $Pr(D_v = t)$ decreases exponentially as t increases, in practice we can numerically estimate the expected decoding time by considering only the first few terms of the sum. In this chapter, we consider j from S_i to $S_i + 3T_d$, which we found to be accurate enough for practical purposes. That is, a packet is considered to be lost at most three times in a row. For larger loss rate, one can consider more terms to trade-off computation time and accuracy.

Our analytical model is useful in several ways. These equations can help us in understanding the effect of dependency (see Section 3.3) when transmitting a progressive mesh over a lossy network. Our model can lead to some simple closed form equations in two special cases. We can also compute the expected decoded quality analytically, leading to a faster alternative to simulation (see Section 3.4.1) as a way to evaluate the effects of network conditions on progressive mesh streaming. Moreover, a better packetization algorithm can be designed based on this analytical model (Section 3.4.2). We introduce these applications of our model next.

3.3 Effects of Dependencies

This section discusses the effects of dependencies on the number of decodable packets, as a function of p and T_d. As mentioned in Section 2.3.2, packetization at the sender may affect decoded mesh quality. We quantify this effect by estimating the number of decodable packets at a given time for two extreme cases of dependencies. The effects of any other dependency will be bounded by what we obtained for these two extreme cases.

In the first case (ideal case), there is no dependency among packets. In the second case (worst case), a packet depends on all other packets sent previously. Note that these two dependency models are independent of any packetization scheme. Moreover, we are computing the number of decodable *packets* rather than *vertex splits*. This simplification does not affect the accuracy of our model since in the ideal case, we can assume that all vertex splits in a packet are decodable as soon as the packet is received (no dependencies among packets). In the worst case, if a packet's dependency is not satisfied, then all vertex splits contained in that packet are not decodable. Thus, the number of decodable packets is proportional to the number of decodable vertex splits in these two extreme cases.

Let $N_{r,t}$ be the number of received packets at time t and $N_{d,t}$ be the number of decoded packets at time t. We show how to compute the expected value of $N_{d,t}$ for the two extreme cases in the next two subsections.

3.3.1 Ideal Case

Theorem 1 *In the case with no dependencies among the packets,*

$$E[N_{r,t}] = E[N_{d,t}] = (t+1)(1-p).$$

Proof In the ideal case, each packet can be decoded independently regardless of other packets. Thus, at any time t, $N_{d,t} = N_{r,t}$. Let a_i represent the result of a transmission at time i, that is

$$a_i = \begin{cases} 1 & \text{if transmission is successful} \\ 0 & \text{if transmission is failed} \end{cases}$$

The expected number of successful transmissions at time t is

$$\begin{aligned} E\big[\sum_{i=0}^{t} a_i\big] &= \sum_{i=0}^{t} E[a_i] \\ &= \sum_{i=0}^{t}(1-p) \\ &= (t+1)(1-p) \end{aligned}$$

□

3.3.2 Worst Case

In the worst case, a packet depends on all previous packets (packets with smaller index). The expected number of packets decodable at time t, is given by the following equation:

$$E[N_{d,t}] = \sum_{i=0}^{m} \big(\prod_{j=0}^{i} Pr(R_j \leq t)\big), \tag{3.3}$$

where m is the total number of packets[2].

[2]When $p = 0$, $N_{d,t}$ is always $t+1$. We assume non-zero p in this section.

Obtaining a close form formula for $E[N_{d,t}]$ is more tricky.

We therefore first derive a close form formula for Δ_t, which is the increase of the expected number of packets decodable at time t, that is,

$$\Delta_t = E[N_{d,t}] - E[N_{d,t-1}].$$

We let Δ_0 be $(1-p)$ since the expected number of packets decodable at time 0 is $(1-p)$.

Suppose P_0 is successfully received at time 0. We remove P_0 from the dependency graph, reducing the dependency graph to a graph with the same dependency structure but with one less packet. The expected number of decodable packets from time 1 to time t is the same as the expected number of decodable packets from time 0 to time $t-1$. Therefore, when $t > 0$, we have

$$E[N_{d,t}|_{P_0 \text{ received}}] - E[N_{d,t-1}] \quad = \quad 1. \qquad (3.4)$$

Now consider what happens if P_0 is lost in its first transmission. For simplicity, we use "P_0 is lost" in the equations when we actually mean P_0 is lost in the first transmission. P_0 will be

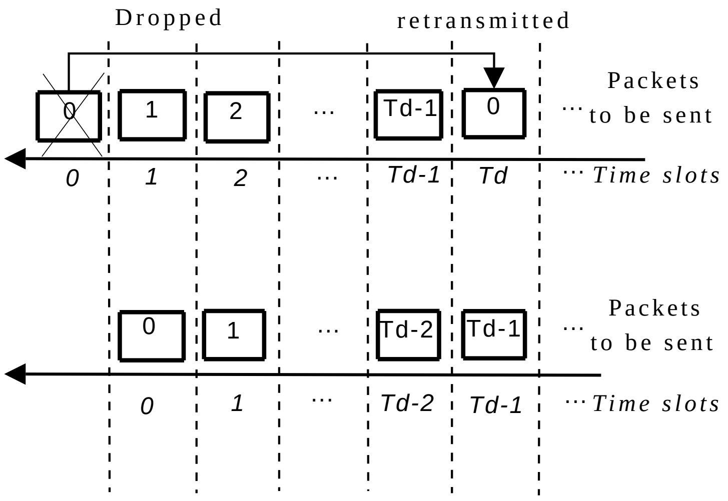


Figure 3.2: If packet 0 is lost in the first transmission, compared with the transmissions begin from time slot 1, only the sending order of first T_d packets is different.

retransmitted at time T_d. If we view time $t = 1$ as the new starting time of transmission, then the only difference between the new sending order and the original one is that in the new sending order, P_0 is sent at time $T_d - 1$ and P_1 to P_{T_d-1} will be sent one slot earlier (see Figure 3.2).

The sending time and receiving time of other packets will be the same. Therefore we have:

$$Pr(R_i \leq t|_{P_0 \text{ is lost}}) = \begin{cases} Pr(R_{T_d-1} \leq t-1) & \text{if } i = 0 \\ Pr(R_{i-1} \leq t-1) & \text{if } 1 \leq i \leq T_d - 1 \\ Pr(R_i \leq t-1) & \text{if } i \geq T_d \end{cases}, \tag{3.5}$$

and hence

$$\prod_{j=0}^{i} Pr(R_j \leq t|_{P_0 \text{ is lost}}) = \prod_{j=0}^{i} Pr(R_j \leq t-1) \quad \text{for any } i \geq T_d. \tag{3.6}$$

In other words, changing sending order of first T_d packets will not affect the decodable probability of later packets.

Moreover, from Lemma 3, we have the following Lemma:

Lemma 4 *At time $t = nT_d$, we have*

$$Pr(R_i \leq t-1) = 1 - p^n \textit{when } 0 \leq i \leq T_d.$$

At time $t = nT_d + b$ and $b > 0$, we have

$$Pr(R_i \leq t-1) = \begin{cases} 1 - p^{n+1} & \textit{when } 0 \leq i < b \\ 1 - p^n & \textit{when } b \leq i < T_d \end{cases}$$

Proof The sending time of packet i $(0 \leq i < T_d)$ is always i (recall that the time unit is the time to transmit a packet). Before time nT_d, they all have n chances to be transmitted, and the probability to be received before time nT_d is $1 - p^n$. At time $nT_d + b$ and $b > 0$. The packet 0 to packet $b-1$ will have $n+1$ chances to be transmitted, therefore the probability to be received before time $nT_d + b$ is $1 - p^{n+1}$. The others have only n chances to be transmitted, and hence the probability to be received before time $nT_d + b$ is $1 - p^n$. □

The above lemma means when t is a multiple of T_d, then all first T_d packets have same probability to be received before or at time t. If $t = nT_d + b$, then the first b packets will have higher probability to be received since they have one more chance to be retransmitted.

Then we can prove the following lemma:

Lemma 5

$$E[N_{d,t}|_{P_0 \textit{ is lost}}] - E[N_{d,t-1}] = \frac{p-1}{p}(1-(1-p^{n+1})^b)$$
$$\textit{where } t = nT_d + b > 0\textit{; } 0 \le b < T_d.$$

Proof

$$\begin{aligned}
&E[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}] \\
=&\sum_{i=0}^{m}\prod_{j=0}^{i} Pr(R_j \le t|_{P_0 \text{ is lost}}) - \sum_{i=0}^{m}\prod_{j=0}^{i} Pr(R_j \le t-1) && \text{(from Eqn. 3.3)} \\
=&\sum_{i=0}^{T_d-1} [\prod_{j=0}^{i} Pr(R_j \le t|_{P_0 \text{ is lost}}) - \prod_{j=0}^{i} Pr(R_j \le t-1)] && \text{(from Eqn. 3.6)} \\
=&[Pr(R_0 \le t|_{P_0 \text{ is lost}}) - Pr(R_0 \le t-1)] \\
&+ \sum_{i=1}^{T_d-1} [\prod_{j=0}^{i} Pr(R_j \le t|_{P_0 \text{ is lost}}) - \prod_{j=0}^{i} Pr(R_j \le t-1)] \\
=&[Pr(R_{T_d-1} \le t-1) - Pr(R_0 \le t-1)] \\
&+ \sum_{i=1}^{T_d-1} \left\{[Pr(R_{T_d-1} \le t-1) - Pr(R_i \le t-1)]\prod_{j=0}^{i-1} Pr(R_j \le t-1)\right\} && \text{(from Eqn. 3.5)}
\end{aligned}$$

We now consider two cases. First, suppose $b = 0$. From Lemma 4, the receiving probability of all packets from 0 to $T_d - 1$ are the same. So

$$[Pr(R_{T_d-1} \le t-1) - Pr(R_i \le t-1)]$$

is always 0 and hence

$$E[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}] = 0.$$

Since $\frac{p-1}{p}(1-(1-p^{n+1})^b) = 0$ when $b = 0$, the lemma holds.

Now, consider the case where $0 < b < T_d$. From Lemma 4, we have

$$\begin{aligned}
&E[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}] \\
=&[1 - p^n - (1 - p^{n+1})] \\
&+ \sum_{i=1}^{b-1} \big\{[1 - p^n - (1 - p^{n+1})] \prod_{j=0}^{i-1} (1 - p^{n+1})\big\} \\
=&(p^{n+1} - p^n)\big[1 + \sum_{i=1}^{b-1} (1 - p^{n+1})^i\big] \\
=&(p^{n+1} - p^n) \sum_{i=0}^{b-1} (1 - p^{n+1})^i \\
=&\frac{p-1}{p}(1 - (1 - p^{n+1})^b)
\end{aligned}$$

Hence, the lemma still holds in this case. □

In Lemma 5, $E[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}]$ is negative if $b > 0$. This case occurrs if packet 0 lost in the first transmission and is sent after packet $T_d - 1$ (See Figure 3.2). Therefore, packet 0 will have less chance to be retransmitted and packet b will have more chance to be retransmitted instead. Since packet 0 has more dependants than packet b, $E[N_{d,t}|_{P_0 \text{ is lost}}]$ is smaller than $E[N_{d,t-1}]$. Moreover, this difference decreases when n increase since the probability for packet 0 and packet b to be lost after n retransmissions is exponentially decreasing.

Combining the results in this section, we have the following:

Lemma 6

$$\Delta_t = (1 - p)(1 - p^{n+1})^b,$$

where $t = nT_d + b$; $0 \le b < T_d$.

Proof

$$\begin{aligned}
\Delta_t &= (1 - p)E[N_{d,t}|_{P_0 \text{ received}}] + pE[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}] \\
&= (1 - p)\{E[N_{d,t}|_{P_0 \text{ received}}] - E[N_{d,t-1}]\} + p\{E[N_{d,t}|_{P_0 \text{ is lost}}] - E[N_{d,t-1}]\}
\end{aligned}$$

when $t > 0$, from Eqn 3.4 and Lemma 5

$$
\begin{aligned}
&= (1-p) + (p-1)(1-(1-p^{n+1})^b) \\
&= (1-p)(1-p^{n+1})^b.
\end{aligned}
$$

when $t = 0$, the above formula still holds because $\Delta_0 = 1 - p$. $\square$

Then, finally we could obtain $E[N_{d,t}]$ as follows:

Theorem 2

$$
\begin{aligned}
\textit{if } t < T_d \textit{ then, } E[N_{d,t}] &= (1-p)\Big[\frac{1-(1-p)^{t+1}}{p}\Big] \\
\textit{else, } E[N_{d,t}] &= (1-p)\Big\{\sum_{i=1}^{n}\Big[\frac{1-(1-p^i)^{T_d}}{p^i}\Big] + \frac{1-(1-p^{n+1})^{b+1}}{p^{n+1}}\Big\}
\end{aligned}
$$

where $t = nT_d + b$; $0 \le b < T_d$.

Proof We have $E[N_{d,t}] = \sum_{i=0}^{t} \Delta_t$, and from Lemma 6, we find that the Δ_t during each T_d is a part of geometric series. From

$$\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q},$$

we can prove this theorem. $\square$

3.3.3 Insights

We have derived the expected decodable number of packets for two cases of dependencies: the ideal case where there are no dependencies between the packets, and the worst case, where a packet always depends on previously sent packets. Any packetization algorithms will lead to a dependency structure between the packets that lies between these two extreme cases. The difference between the number of decodable packets for these two cases therefore gives us an indication of how much improvements we can get if we intelligently group the vertex splits into packets.

The next question is how big is the gap between the two extreme cases. Let $f(i)$ be the difference of average decodable number of packets in time iT_d. Then from Theorem 1 and

Theorem 2, we have

$$\begin{aligned} f(1) &= (1-p)T_d - (1-p)\frac{1}{p}[1-(1-p)^{T_d}] \\ f(i)-f(i-1) &= (1-p)T_d - (1-p)\frac{1}{p^i}[1-(1-p^i)^{T_d}], \end{aligned}$$

where $i \geq 1$. Considering

$$\begin{aligned} & (1-p)\frac{1}{p^i}[1-(1-p^i)^{T_d}] \\ = \quad & -(1-p)\frac{1}{p^i}\sum_{j=1}^{T_d}\left[(-1)^j\binom{T_d}{j}p^{ij}\right] \\ = \quad & -(1-p)\sum_{j=1}^{T_d}\left[(-1)^j\binom{T_d}{j}p^{i(j-1)}\right] \\ = \quad & (1-p)T_d - (1-p)\sum_{j=2}^{T_d}\left[(-1)^j\binom{T_d}{j}p^{i(j-1)}\right], \end{aligned}$$

we have

$$\begin{aligned} f(i)-f(i-1) &= (1-p)\sum_{j=2}^{T_d}\left[(-1)^j\binom{T_d}{j}p^{(j-1)i}\right]; \\ f(i) &= (1-p)\sum_{j=2}^{T_d}\left[(-1)^j\binom{T_d}{j}\sum_{k=1}^{i}p^{(j-1)k}\right]; \\ f(\infty) &= (1-p)\sum_{j=2}^{T_d}\left[(-1)^j\binom{T_d}{j}\frac{p^{j-1}}{1-p^{j-1}}\right] \\ &\leq \sum_{j=2}^{T_d}\left[\binom{T_d}{j}\frac{(-p)^j}{p}\right] \\ &= \frac{1}{p}[(1-p)^{T_d}-1+T_dp] \end{aligned}$$

Therefore, we can see the difference is upper-bounded. Since when p is small, $\sum_{k=1}^{i} p^k$ converges to $\frac{p}{1-p}$ quickly, so the difference after 2 to 3 T_d is already very close to the upper-bound value.

The main observation from our analysis is that *dependencies matters only during the first few T_ds.* In Theorem 2, if t is large, and p is small enough, then the factor $1-p^{n+1}$ tends to 1. Hence, as time progresses, the increase in number of decodable packets for the worst case becomes close to $1-p$ for each time slot. If we plot two curves, showing expected number of decodable packets versus time, for the ideal and worst case, then the two curves become almost parallel for large values of t. In fact, if we consider the difference between the curve at time xT_d, then the difference tends to a constant that depends only on T_d and p as x tends to infinity. Figure 3.3 shows an example of such two curves, with $p = 0.1$ and $T_d = 40$.

This observation leads us to believe that optimization of packet dependencies only matters during the first few T_ds. After that, the dependencies among the packets will not affect the number of decodable packets. Further, in progressive mesh, the importance of the vertex splits decreases as the model becomes incrementally refined. Thus, the contributions of the later vertex splits to the decoded quality of the mesh are less than the contribution of previous vertex splits.

This observation is good news – regardless of how large the progressive mesh is, only dependencies among vertex splits sent during the first few T_ds matter. Thus, any packetization algorithm only needs to focus on the vertex splits sent during this initial period, reducing the computational costs significantly.

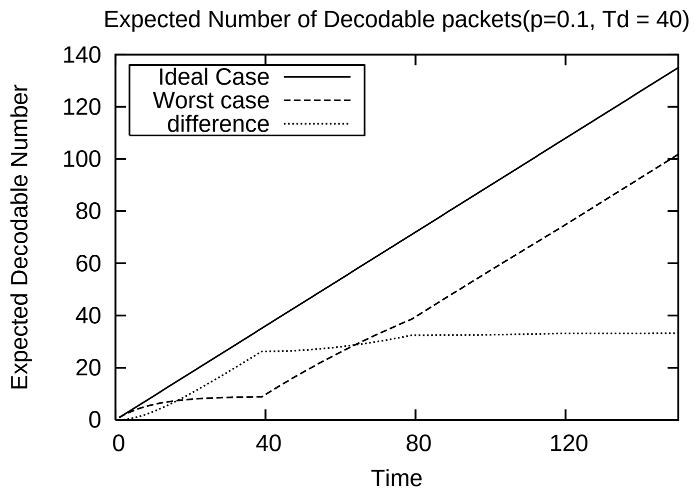


Figure 3.3: The number of decodable packets for ideal case and worst case. $T_d = 40$, Loss rate $p = 0.1$. In the figure, we assume the transmission begins at time 1 so all curves begin from the origin.

3.4 Improving Mesh Quality

Using our analytical model, we can now analytically compute the expected decoded mesh quality given the dependencies among vertex splits and the sending order of the packets. We first explain how we measure the quality of a progressive mesh.

3.4.1 Expected Decoded Mesh Quality

The quality of a simplified mesh represents how close it is compared to the original mesh. Some objective metrics have been proposed. Many of them are based on the Hausdorff distance between a set of sample points on the original mesh and the corresponding ones on the simplified one. For example, Metro [26] and M.E.S.H [8] can be used to obtain both the mean and the maximum face to face distances between two models.

In the case of a progressive mesh, the base mesh has the lowest quality, and each vertex split increases the quality by a certain amount. We define the *importance* of a vertex split as the increase in decoded mesh quality due to this vertex split. This value can be determined by comparing the quality of the model before and after the vertex split operation. Strictly speaking, the importance value may depend on the split order, but, for simplicity, we assume that a refinement always improves the quality of the model by a value, independently of other refinements. Then, the quality of a received model at time t (or, *intermediate quality*) can be represented as the summation of importance of all vertex splits decodable at t. Since the simplification process typically prefers collapsing an edge with low quality loss, edges with lower importance are collapsed earlier during the simplification and split later during the reconstruction. Therefore, in a progressive mesh, vertex splits operations are typically performed in decreasing order of importance.

This model of decoded mesh quality is general – one can define different importance metric for a vertex split depending on how mesh quality is measured. For instance, if the mesh quality is view dependent, one can set the importance of a vertex split to zero if the vertex split is outside of the user's viewpoint [40].

In progressive streaming, the quality of the mesh on the receiver increases over time, and plotting this quality versus time gives us a *quality curve*. We evaluate and compare different streaming strategies of the same progressive mesh by comparing the quality curve. Because vertex splits contribute differently to the quality, the quality curve depends on the decoding time and decoding order of the vertex splits.

Now we explain how our model can be used to estimate the quality curve. The quality at time t is the sum of importance of decoded vertex splits. That is

$$q_t = \sum_i w_i, \text{for all vertex split } i \text{ decoded before or at time } t.$$

We define s_i such that $s_i = 0$ if vertex split i is not decoded yet at time t, and $s_i = 1$ if vertex split i is already decoded at time t. Suppose we have n vertex splits in total. Then we have $q_t = \sum_{i=0}^{n} s_i w_i$, so $E[q_t] = \sum_{i=0}^{n} w_i E[s_i] = \sum_{i=0}^{n} w_i Pr(D_i \leq t)$. Since $Pr(D_i \leq t)$ can be obtained from Equation 3.1, we are able to compute $E[q_t]$. Therefore, we can evaluate a sending order by predicting the expected quality curve with our analytical model given the network condition and the mesh property. Moreover, in next section, we introduce that the predicted quality can also be used in designing streaming strategies, particularly, packetization of vertex splits.

3.4.2 A Packetization Algorithm

Packetization refers to the process of grouping vertex splits into packets before transmitting them over the network. Due to dependencies among nodes, packetization may introduce dependencies among packets (if a vertex split and its parent are packed in different packets). Therefore dependencies affect the decoding time of the vertex splits and the intermediate quality of the decoded model. The intermediate quality of the rendered model can be important in interactive applications, especially during the initial stage of streaming, since users may need to react to what they see quickly.

To improve the intermediate quality of the decoded model, we need to consider two factors in the packetization: the importance of each vertex split, or node, and the dependencies among nodes. One strategy is to always send the most important node first; the other is to packetize the packets to minimize its dependency [35]. If there is no packet loss, these two objectives can be achieved at the same time. Since the vertex split operations in a progressive mesh are typically executed in the decreasing order, we can simply send the vertex splits in the first-in-first-out (FIFO) order, in the reverse order of the edge collapse operations. Since a node's parent packets are always sent before the packet containing the node, a node can be decoded as soon as it is received. When packet loss exists, however, there is a conflict between maximizing importance and minimizing dependencies. FIFO satisfies the first objective, but may increase dependency among packets; whereas to reduce dependency, one may send a node with lower importance before a more important node. To trade-off between these two objectives, we need to know the exact effect of both factors. In this section, we show how we use our model developed in Section 3.2 to help us select which node to pack.

Before introducing our method called greedy, we need to quantify the quality at a given time t. Simply comparing the quality at time t is too restrictive. For instance, in Figure 3.4(a), although the quality at time t is the same for both strategies, we think that Strategy 2 is better since it can achieve a higher quality earlier.

Based on this observation, we propose an evaluation metric that measures the intermediate quality over a period of time (rather than instantaneous). The metric sums up the intermediate quality of the mesh over a given time period. Imagine plotting the quality of the received mesh versus time, as in Figure 3.4. This metric is equivalent to the area between the curve and the time axis. We can interpret the meaning of this metric in two ways . We view the metric as the sum of decoded quality from time 0 to t. Discretizing the time, we let q_t be the decoded quality of the progressive mesh at time t, and let a_t be the area under the curve at time t. Then, $a_t = \sum_{i=0}^{t} q_i$. Here, the area under the curve is computed as the area sum of vertical slices (see Figure 3.4(b)). We can also compute the area as the area sum of horizontal slices (see Figure 3.4(c)). Thus, to compute the area, we can sum up the product of a vertex split's importance and the amount of time since it was decoded. Let the importance of a vertex split v be w_v and the time when it was decoded be D_v. Then

$$a_t = \sum_{v \in K_t} w_v (t - D_v), \tag{3.7}$$

where K_t is the set of vertex splits that have been decoded at time t.

Consider a node v. We need to decide whether we should pack v into the current packet. First, note that if there exists a parent of v that has not been packed, then we should not have packed v (if a parent of v arrives later than v, v cannot be decoded anyway). Thus, we only consider nodes whose parents have all been packed. Now, consider what would happened if we

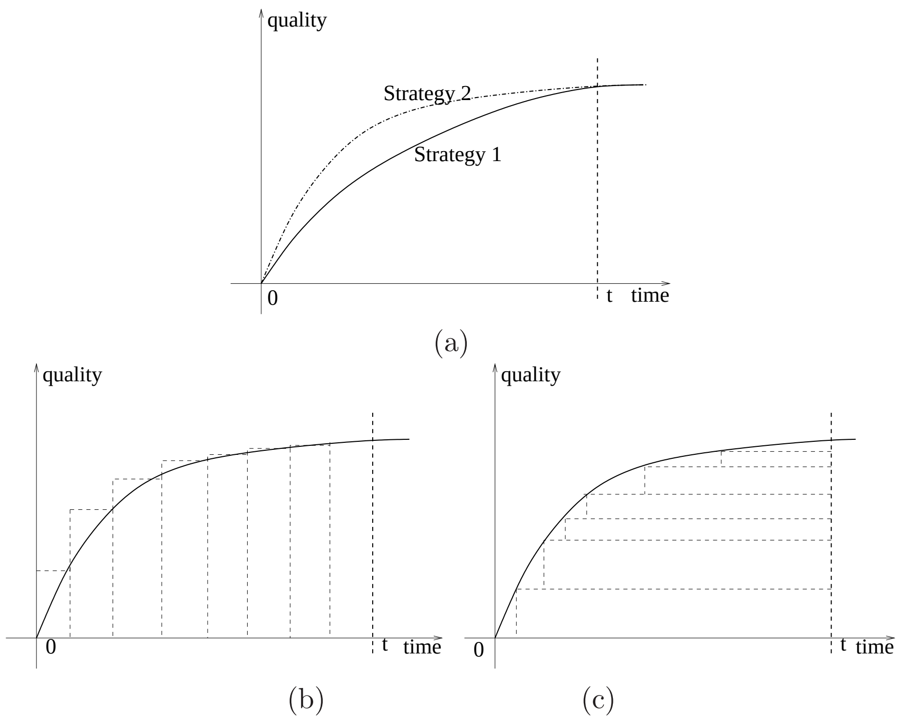


Figure 3.4: Intermediate quality of decoded mesh (imaginary). From left to right: (a) Strategy 2 is better than Strategy 1 since the area under the curve is larger. (b) Area sum of vertical slices. (c) Area sum of horizontal slices.

pack v into the current packet, versus the subsequent packet. From Equation 3.7, the difference in quality, δ_v, between the two cases is given by

$$\delta_v = w_v(E[D_v^{next}] - E[D_v^{curr}]), \tag{3.8}$$

where D_v^{curr} and D_v^{next} are the decoding time of v if v is packed in the current packet and next packet respectively. We call the metric δ_v as the *penalty.* To compute the penalty, we use Equation 3.2 from our model, which gives us the expected decoding time for these two cases. Minimizing the penalty maximizes the difference in decoded mesh quality (Equation 3.7).

Equation 3.8 succinctly captures the trade-off between the importance of v and the dependencies. The penalty increases as w_v increases. Consider the case where v has a parent in the current packet. If we pack v in the next packet, then the (expected) decoding time for v increases – not only because it will arrive later, but also because this packing introduces a dependency between the current and the next packet. From Equation 3.2, we can see that additional dependencies increase the decoding time since both packets have to be received before v can be decoded. Therefore, increasing dependencies increases the penalty.

We can now describe a greedy algorithm to packetize progressive meshes. The algorithm simply packs the node with highest penalty at each step[3]. Algorithm 1 shows the pseudo-code for packing one packet using our algorithm.

Algorithm 1 Greedy Packetization

```
for all node $v$ whose parents are already packed do
  calculate its penalty $\delta_v$ if it is moved to the next packet;
  insert $v$ into a maximum heap $H$ with $\delta_v$ as key;
end for
while $H$ is not empty and packet is not full do
  Pop $j$ from $H$;
  Pack $j$ into current packet;
  for all children $k$ of $j$ whose parents are already packed do
    calculate $\delta_k$ if $k$ is moved to the next packet;
    insert $k$ into $H$ with $\delta_k$ as key;
  end for
end while
```

To reduce the computation cost, we approximated δ_v by computing $E[D_v^{next}]$ and $E[D_v^{curr}]$ up to a limited number of terms. We chose to use up to $3T_d$ terms in our implementation. Further, from our observation in Section 3.3, since the effect of dependencies diminishes with time, we stop running the algorithm after time $3T_d$ and simply send the vertex splits in decreasing order of their importance. We evaluate the effectiveness of our proposed greedy packetization in Section 3.5.4.

3.5 Model Validation

In deriving our analytical model, we made several simplification to make the model tractable. We use the average packet loss rate, sending rate, and round trip time (RTT) in modeling the network conditions. When we model the mesh, we assume that the importance of a vertex split is independent of the order and is additive. In this section, we validate the accuracy of our analytical model to show that, despite the simplifications, our model is still accurate (i) under realistic network conditions, (ii) when using a non-additive, order-dependent quality metric to represent the importance of a vertex split, and (iii) using different progressive meshes.

[3]Technically this is a heuristic since it does not guarantee an optimal packetization. The packetization problem has been shown to be NP-complete [35].

3.5.1 Experimental Setup

Network Traces For reproducibility, we recorded packet traces of a lengthy transmission between Singapore and Toulouse over the Internet during the summer of 2008 and simulate mesh transmissions using the recorded traces. The traces allow us to validate the predicted expected values against the averages from multiple runs under the same network conditions. We use both UDP and DCCP CCID2 [50]. UDP packets are sent at a constant rate as required by our analytical model. However, losses are bursty and packet loss rate fluctuates over time. DCCP transmissions give variable sending rate (due to congestion control) but experiences very few losses. DCCP also implemented a TFRC-based congestion control mechanism (CCID3), but we found the implementation buggy as of Linux kernel 2.6.24. CCID3, however, generates a smoother sending rate than CCID2, so if our model works for CCID2, it will work even better for CCID3.

Possibly due to firewall or backbone routers, DCCP does not work between Toulouse and Singapore, so we implemented a UDP tunnel that encapsulates DCCP packets. Tunnelling with UDP allows us to preserve the loss rate and the RTT of the underlying network. At the sender, our tunnel captures the outgoing packets (using `libpcap` and filtering), encapsulates them in UDP packets and transmits the packets. At the receiver, the tunnel decapsulates the UDP packet and sends the DCCP packet through a raw socket to the DCCP receiver. Moreover, we use our tunnel to *spy* on the DCCP protocol by decoding the packet headers. We can therefore use DCCP's acknowledgement mechanism to compute the RTTs (acknowledgements are, otherwise, not visible from user-level applications).

From the large set of collected traces we chose three main traces for our validation: (i) UDP-Low – a UDP trace with low loss rate ; (ii) UDP-High – a UDP trace with high loss rate, observed during the 2008 Olympic Games; and (iii) DCCP – a DCCP CCID2 trace. We plot the loss rate, RTT, and throughput of a 90-second segment from each of the three traces in Figure 3.5. Table 3.2 shows the main characteristics of the traces.

Trace	Length (second)	Loss Rate p (%)	Sending Rate (kbps)	RTT (ms)	T_d
UDP-High	537.1	14.53	2,087	412	77
UDP-Low	85.2	3.6	13,204	717	845
DCCP	460.3	0.085	488	660	29

Table 3.2: Main characteristics of the transmission traces.

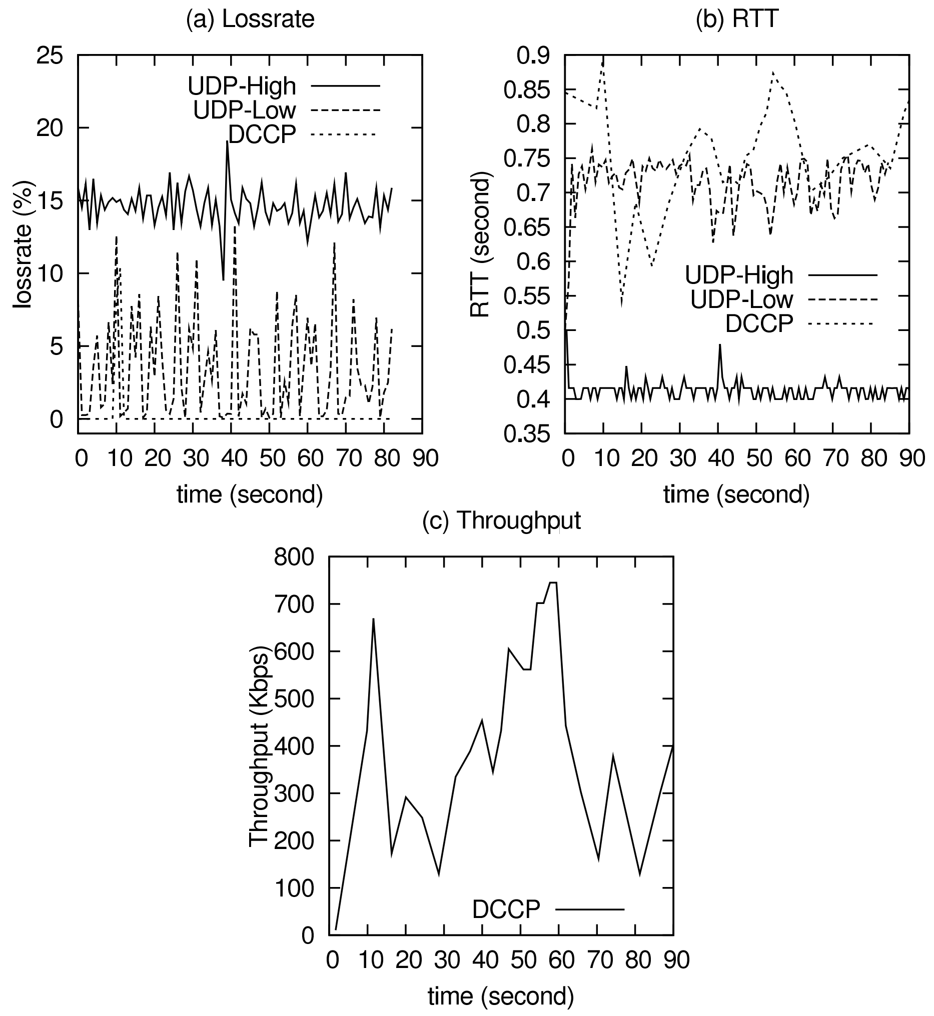


Figure 3.5: Loss rate, RTT, and throughput of our traces

.

Meshes We chose three meshes with different characteristics and complexity to validate our model. The meshes we picked are *Horse*, (from Georgia Tech[4]), *Happy Buddha*, and *Thai Statue* (from Stanford 3D scanning repository[5]). Characteristics of these meshes are listed in Table 3.3.

3.5.2 Validation of Receiving Time and Decoding Time

To validate the accuracy of our prediction of receiving time of each packet $E[R_i]$ and decoding time of each vertex split, $E[D_v]$, we measure the actual receiving time and decoding time of

[4]http://www.cc.gatech.edu/data_files/large_models/horse.ply.gz

[5]http://www-graphics.stanford.edu/data/3Dscanrep. The *Thai Statue* we used is a simplified version of the original one.

Mesh	Vertices (total)	Faces (total)	Vertices (base)	Faces (base)	Base Mesh KBytes	Vertex Splits MBytes
Horse	48485	96966	227	450	8.14	1.567
Happy Buddha	543652	1631574	4703	9818	174.268	19.565
Thai Statue	1000024	2000056	610	1228	22.072	36.545

Table 3.3: The characteristics of the different meshes.

vertices during transmission of our meshes. Note that the receiving time of a vertex is just the receiving time of the packet that contains this vertex. The measured values are then compared to the expected value predicted by our model. We detail this validation process in this section.

Different Network Traces To compute the expected value of receiving time and decoding time from our model, we plug in the average p and T_d for different traces from Table 3.2. To obtain the receiving time and decoding time from the packet traces, we simulate the mesh transmission and log the receiving time and decoding time of each vertex. For each trace, we transmit the mesh 10 times (i.e., 10 runs) and in each run, we randomly pick a time in the trace as the starting point. We average the value of receiving time and decoding time over 10 runs and plot the measured average values along with the predicted values from our model (using the mesh *Happy Buddha*) in Figure 3.6.

Figure 3.6 shows that the predicted values based on our model is reasonably close to the measured values from the experiments in all three traces. The predicted receiving time is more accurate than the predicted decoding time since the decoding time of a vertex split depends on the receiving time of all its parent packets, thus an error in the receiving time propagates to the prediction of decoding time. The result of DCCP trace shows that despite large variation in sending rate, our model, which assumes constant sending rate, still works well.

Figure 3.6 also shows that the predicted decoding time for trace UDP-Low is considerably larger than the measured value in the first T_d. This discrepancy is caused by many bursty packet losses in trace UDP-Low. A vertex split can be decoded only after *all* its parent packets are received. For vertex splits sent in the first T_d, their parent packets are typically sent close to each other. Compared to traces with independent packet losses, bursty traces with the same average loss rate have higher chances of having long continuous sequence of successful transmissions. Hence, a vertex split with many parent packets sent in a short period is more likely to have all its parent packets successfully received, leading to smaller average decoding time. On the

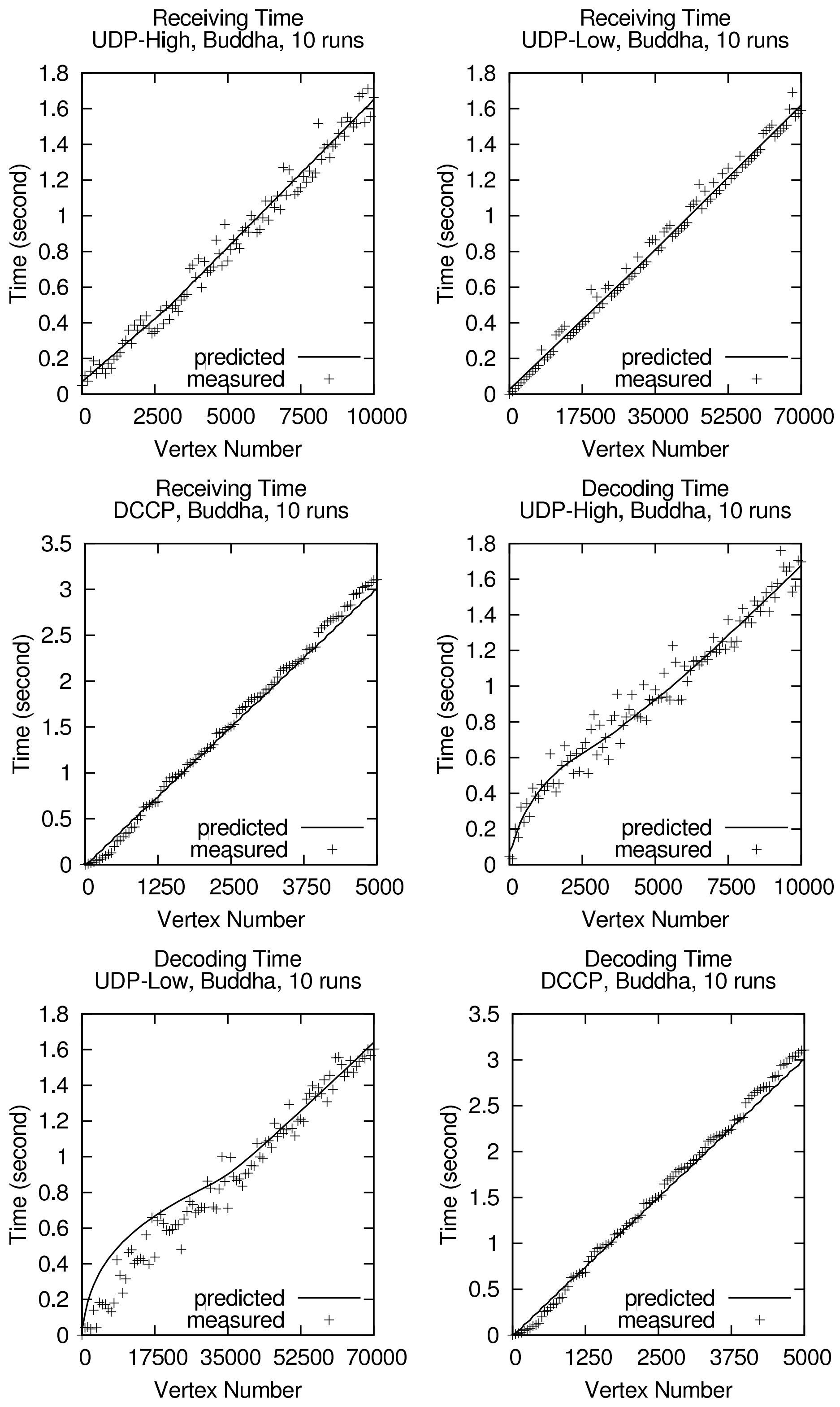


Figure 3.6: Comparing the values of receiving time $E[R_v]$ and decoding time $E[D_v]$ as predicted by our model and as measured from our experiments, using the *Happy Buddha* mesh averaged over 10 runs.

other hand, our model assumes independent packet losses. With a more evenly distributed loss pattern, a vertex split has a higher chance of losing one or more of its parent packets, hence leading to larger predicted decoding time. This discrepancy becomes negligible for the vertex splits sent after the first T_d because their parent packets have chances to be retransmitted, and, moreover, their parent packets are sent further apart compared with the parent packets of the vertex splits sent in the first T_d.

Different Meshes We repeat the comparisons above using two other meshes *Horse* and *Thai Statue*. We present the results for UDP-High in Figures 3.7. Results for other traces are similar and are therefore omitted.

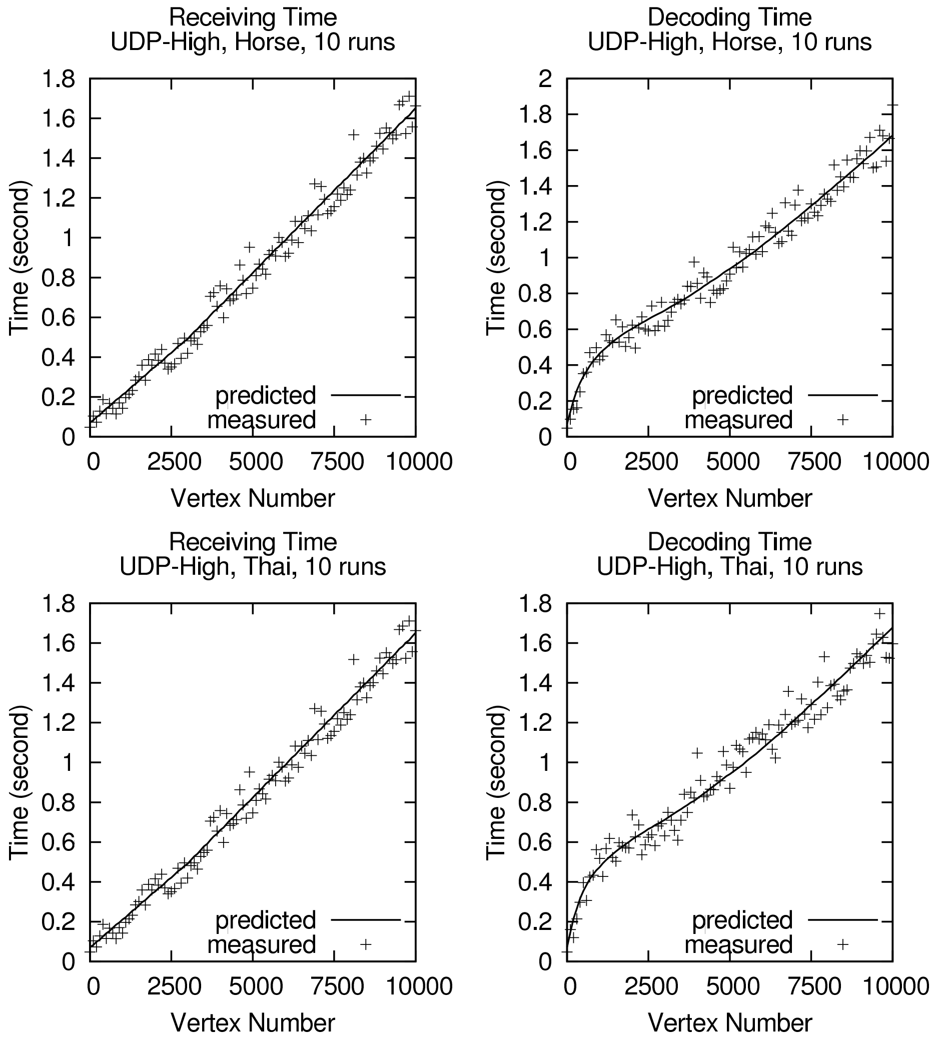


Figure 3.7: Comparing the values of receiving time $E[R_i]$ and decoding time $E[D_i]$ as predicted by our model and as measured from our experiments, using various meshes.

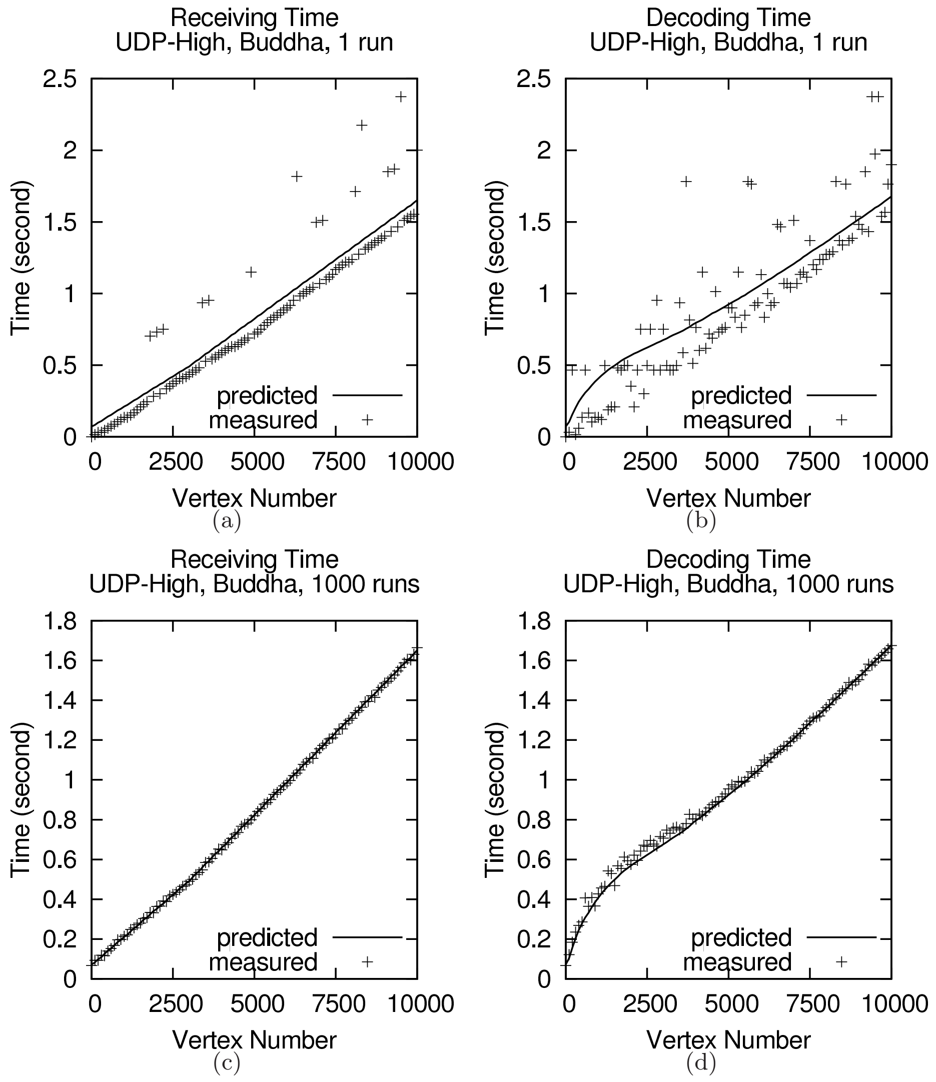


Figure 3.8: Comparing the values of receiving time $E[R_i]$ and decoding time $E[D_i]$ as predicted by our model and as measured from our experiments, using various meshes and number of runs.

Single Run Figures 3.8(a)-(b) shows the predicted values of receiving time and decoding time compared to the measured ones for a single run. We can see that most measured values are slightly smaller than the predicted ones, while some measured values are considerably larger than the predicted ones. For receiving time, larger values are observed when the corresponding vertex splits is lost. For decoding time, the larger measured values corresponds to cases when a vertex split or its parents packets are lost.

Large Number of Runs Figures 3.8(c)-(d) compares the predicted values of receiving time and decoding time to the measured ones, averaged over 1000 runs. The close match of the results shows that our model accurately predicts the expected value of receiving time and decoding time.

3.5.3 Validation of Quality Curve

We now validate the accuracy of the quality curve. First, we compare the predicted mesh quality with the measured quality under different traces and different meshes. For these experiments, we use the length of the collapsed edge as the importance of the vertex split. We use this metric because (i) it is used by the library we use in our experimental testbed (GTS [6]), and (ii) it is strictly independent of the decoding order. We then compare the quality curve measured using split edge length with another commonly accepted metric, the mean face-to-face Hausdorff distance, to show that our model are still accurate when a different metric is used.

Different Traces First, we compare the predicted quality curve from our model and the measured quality curve, averaged over 10 runs, using the three traces. We can see that the two curves fit reasonably well in all cases, except the first T_d for Trace UDP-Low due to the high bursty packet loss. We have explained that the decoding time of vertices sent in the first T_d tends to be smaller than predicted value due to the high bursty packet loss. As a result, the measured quality curve is higher in the first T_d than the predicted value. At multiples of T_d, we can see a sudden increase in the quality curve when we validate using the UDP traces. The increase is explained by Lemma 6. When t is a multiple of T_d, the number of decodable packets increases by $1-p$, which is significantly larger than the previous increase $(1-p)(1-p^{n+1})^{T_d-1}$ (when n is small and T_d is large). For DCCP, however, packet losses are rare, and thus, the number of decodable packets increases smoothly.

Different Meshes Next, we verify the accuracy of our model using different meshes. Using the trace UDP-High, we repeat the experiments for two other meshes, *Horse* and *Thai Statue*. The result is shown in Figure 3.10. Again, the predicted and measured curves are close together, indicating that our model can be applied to other meshes.

Different Metrics Our model assumes that the quality metric is additive and is independent of the decoding order. In the validation above, we use the split edge length, which satisfies both

[6]www.gts.org

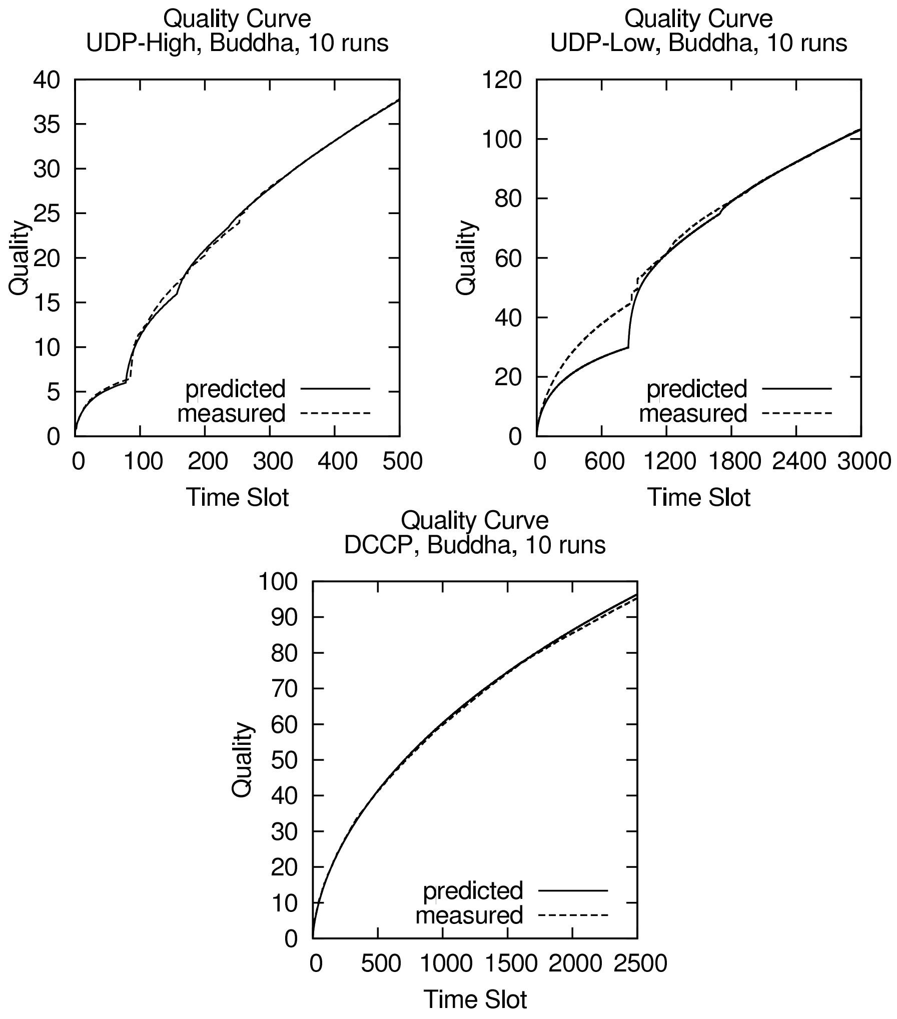
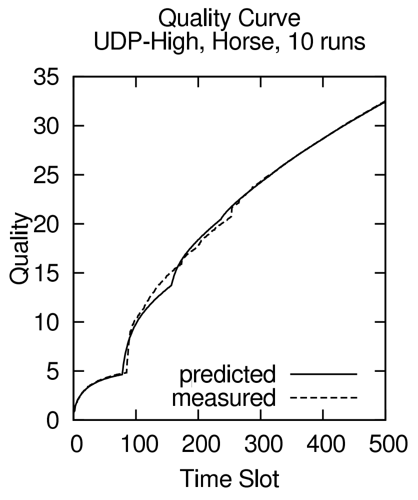


Figure 3.9: Comparing the quality curve as predicted by our model and as measured from our experiments, using three different traces.

properties, as the metric. To verify that our model is accurate under different metrics, we plot the quality curve with another commonly accepted metric – the Hausdorff distance between the reconstructed mesh and the original mesh. We use the difference of mean face-to-face Hausdorff distance between these two meshes before and after split as the importance of a vertex split. This metric is neither additive, nor order-independent. We use M.E.S.H [8] to compute this mean face-to-face distance.

Figure 3.11 shows the quality curve plotted with Hausdorff distance, using the UDP-High trace on *Happy Buddha*. The Hausdorff distance-based metric depends on the decoding order, which depends on packet loss patterns. To obtain the quality curve, we re-compute the Hausdorff distance after every packet for each experiment, and compute the average curve over 10 runs.



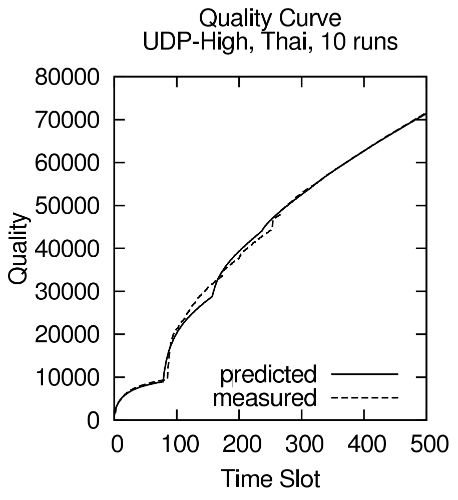


Figure 3.10: Comparing the quality curve as predicted by our model and as measured from our experiments, using *Horse* and *Thai Statue*.

The results shows that, in the first T_d, the curve for Hausdorff distance is slightly higher than the predicted curve from our model. This difference is due to the change in the decoding order caused by packet losses. Since the decrease in Hausdorff distance is often larger when a vertex split is decoded earlier, the importance becomes higher than what we predicted. The two curves, however, are still close to each other, indicating that our model is reasonably accurate even if we use a quality metric that is not additive and depends on the decoding order.

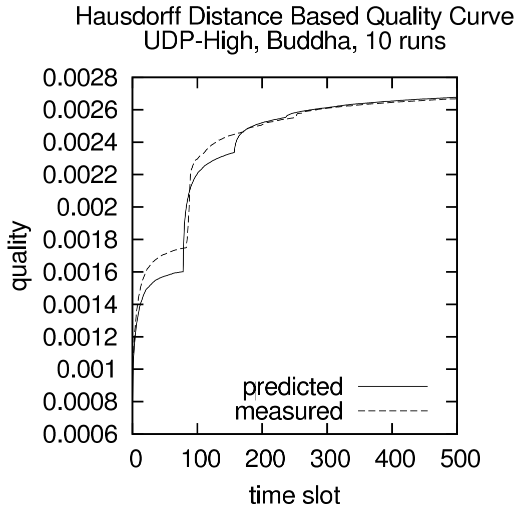
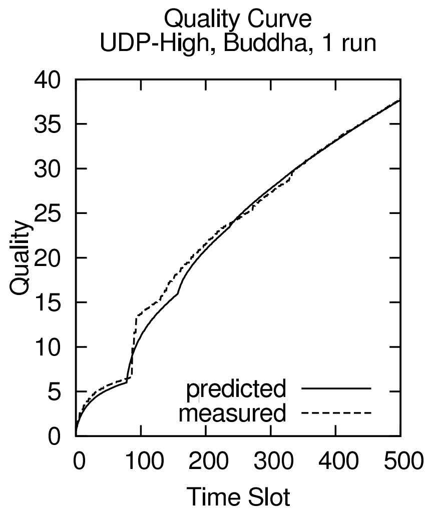


Figure 3.11: Predicted quality curve and measured quality curve using Hausdorff distance as quality metric, averaged over 10 runs.

Single Run We also validate the accuracy of the quality curve from a single run. We pick two sample runs here. The predicted curve is higher than the measured curve for one run and is



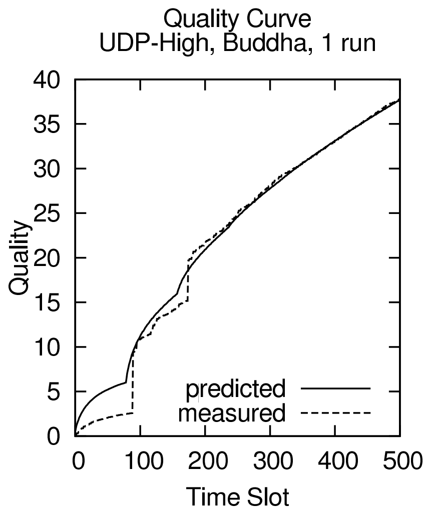


Figure 3.12: Predicted quality curve and measured quality curve from two different single runs.

lower for the other (see Figure 3.12). The measured quality is lower for the second sample since it suffers from some early packet losses. Our model, however, still matches the overall shape of the curve fairly well.

3.5.4 Validation of the Greedy Method

We compare the relative improvement of the greedy algorithm we proposed (Algorithm 1) over the FIFO algorithm on the *Happy Buddha* mesh with two UDP traces. We defined the relative improvement as $(g - f)/f$, where g is the quality of the mesh if transmitted using the greedy method, and f is the quality of the mesh if transmitted using the FIFO method. The results, shown in Figure 3.13, shows that our greedy method outperforms the FIFO in the first several round trip times. The results for the DCCP trace are omitted because this trace has negligible packet losses, and thus, the two algorithms works equally well.

The largest gap in the average quality between FIFO and greedy occurs at time slot 85. The average qualities for FIFO and greedy at this time slot are 5.96 and 9.29 respectively. To visualize the difference, the rendered images of *Happy Buddha* mesh with these qualities are shown in Figure 3.14.

Note that we choose the edge length of a vertex split as its importance. The Hausdorff distance-based metric is not suitable here since it depends on the decoding order of the vertex splits. Changing the sending order changes the importance of vertex splits, so our greedy method is not applicable.

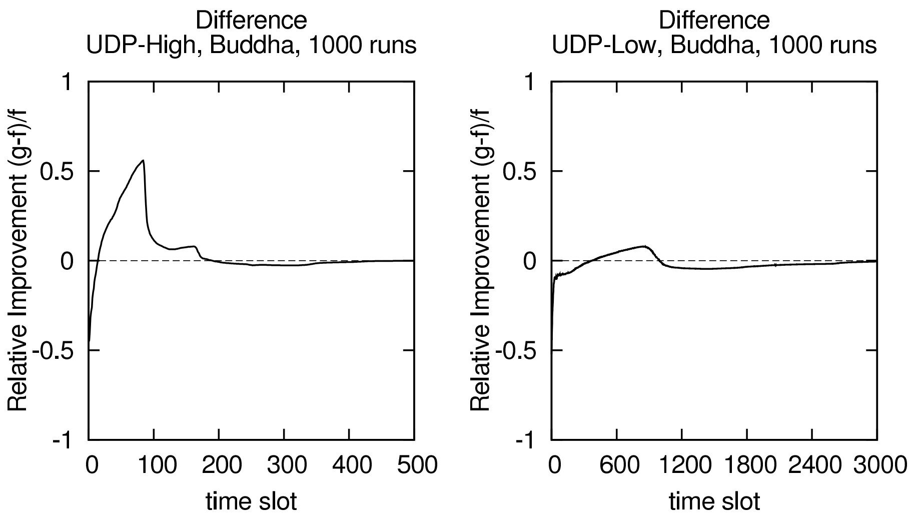


Figure 3.13: The comparison of quality curves for FIFO and greedy.

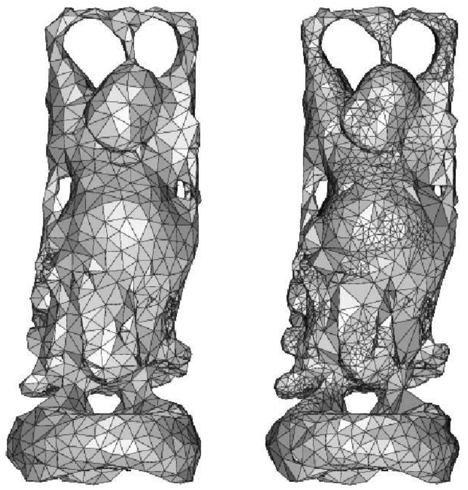


Figure 3.14: The rendered *Happy Buddha* at the time T_d, using UDP-High. Left: FIFO with quality 5.96, Right: greedy with quality 9.29.

3.6 Conclusion

We first conclude by reflecting on what we learn from our model.

The most important insight we gain is that *the effect of dependencies among vertex splits on decoded mesh quality when transmitted over a lossy network is limited to the first few T_ds.* This has several implications. Since T_d is typically small, the effect of dependencies is not going to matter for non-interactive applications. Thus, we believe that existing research to reduce dependencies (for instance, mesh segmentation [65, 84] and packetization [35]) is not relevant in this context if retransmission is used.

For interactive applications, the decoded mesh quality in the first few T_ds is crucial. We showed that FIFO ordering gives good enough average quality – even though a more intelligent greedy packetization can give better average quality.

Besides packetization, our insight that only first T_ds matters can lead to other transmission schemes. For example, the sender can protect the vertex splits sent during the first few T_ds using enough FEC coding, ensuring that there is no losses; or the sender can increase the sending rate temporarily during this brief period to improve the initial quality. Our model is still useful here. For instance, sending FEC would decrease the loss probability of some packets but would delay transmission of new data. Temporarily increasing the sending rate may also increase the loss rate due to congestion. Our model can characterize this trade-off and thus can guide the sender in deciding whether to send a FEC packet or new data. Our formula for expected decodable quality can similarly guide the sender in deciding how fast it should send during the first T_d period.

To show that the insights from the theoretical model can be applied in practice, we extensively verified our model under different conditions, using different progressive meshes, network conditions, and quality metrics. The experimental results show that our model works well under realistic conditions despite the simplification and assumptions we made during modeling.

Three extensions to our model are possible. First, our model could be used in view-dependent streaming by using view-dependent quality metric, such as the contribution of vertex splits to the quality of the screen image [21]. Second, our analysis could be extended to deformable meshes, where the contributions of vertex splits change with user interactions. In both extensions, the contributions of vertex splits change dynamically, requiring the sending order to be computed online. Recomputing the sending order online efficiently remains a challenge. Finally, it would be interesting to extend our model to mesh animation. Mesh animations can still be modeled using directed acyclic graphs, but introduce playback deadlines on vertex splits. This temporal component is not considered in our model. Extending our work to model a mesh animation remains open.

Chapter 4

Receiver-Driven View-Dependent Streaming of Progressive Meshes

4.1 Introduction

During streaming of a progressive mesh, it is desirable to increase the visual quality of the reconstructed mesh on the client side as quickly as possible. Therefore, vertex splits having high contribution to the visual quality should be sent early. Because what users really see is the rendered image on the screen, and the quality of the rendered image depends on not only the quality of the reconstructed mesh but also the viewpoint of the user, the contribution of a vertex split also depends on the viewpoint of the user. As a result, using quality metrics independent of viewpoint, such as the commonly used Hausdorff distance between the original and reconstructed mesh [26], to decide the streaming order, is not accurate enough. Bandwidth may be wasted in sending invisible vertex splits before the visible ones. Moreover, even among the visible vertex splits, the view-independent metric cannot reflect the real contribution to the visual quality of rendered images in clients with different viewpoints. A vertex split that significantly changes a mesh may change the rendered image only slightly.

A better metric of the visual contribution of a vertex split, which considers the receiver's viewpoint, is based on the quality of the rendered image on the screen. An example is the one proposed by Lindstrom and Turk [55]. Based on this kind of view dependent metric,

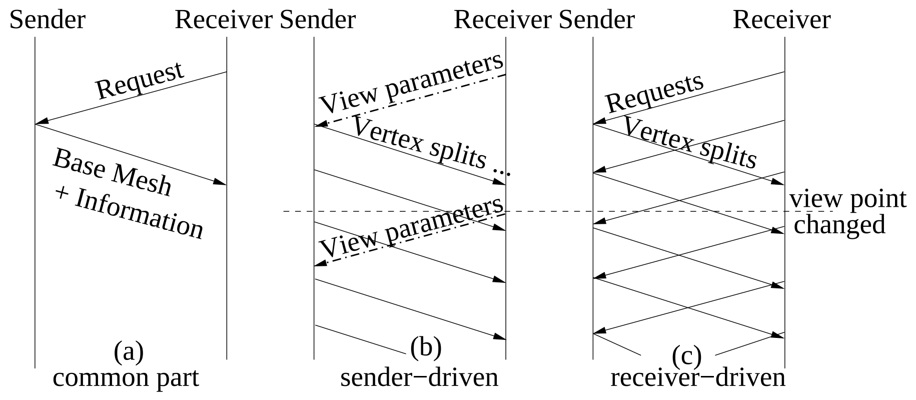


Figure 4.1: Sender-driven protocol and receiver-driven protocol

view-dependent streaming, in which vertex splits are sent in the descending order of their contributions to the quality of rendered image, is introduced to improve the user experience.

In previous implementations of view dependent streaming [79, 73, 85, 48, 91], the sender decides which vertex splits to send. In these implementations, the receiver sends its viewing parameters to the sender, and the sender sends the chosen vertex splits after determining the visibility and the appropriate resolution of different regions of the mesh (See Figure 4.1b).

This sender-driven protocol is not scalable to a large number of receivers (unless using more servers as senders, which significantly increases the infrastructural cost) due to its two characteristics: (i), the sender is state-ful because it needs to remember which vertices are sent for each receiver to avoid sending duplicate data; (ii), the sender has to do expensive computation to determine the visibility and visual contributions of vertex splits for each receiver.

Due to the state-ful design and huge computational requirements, the sender-driven approach cannot be extended easily to support caching proxy and peer-to-peer architecture, two common solutions to scalability. First, it is difficult for a receiver to request data from multiple proxies or peers without receiving duplicate data, because it requires the sending history in multiple senders to be synchronized. Second, it is not realistic to require each proxy or peer to provide much CPU time and memory (maintaining state of other receivers) to other receivers. Third, a proxy or peer might not store the complete mesh for visibility determination.

To address the above weaknesses and design a more scalable view dependent streaming system, we propose a receiver-driven protocol, in which the receiver decides the sending order and explicitly requests the vertex splits. The sender simply sends the data requested (See Figure

4.1c), so no expensive computation is needed. Furthermore, the server is stateless, so existing cache proxy and peer-to-peer techniques can be applied.

Implementing the receiver-driven protocol is non-trivial. First, the receiver has to efficiently determine the visibility and visual contribution of a vertex split based on the partially received mesh. Second, we need to assign each vertex a unique identification number so that the receiver can explicitly request vertex splits. To avoid sending extra data, the identification number of each vertex should be implicitly known by the receiver. Third, most current coding and compression methods exploit the sending order of vertex splits to increase the compression efficiency as we introduced in Chapter 2. We now, however, need a new compression scheme completely independent of the sending order since the sender has no freedom in choosing sending order and even does not remember the sending order.

We introduce the problems with more details in Section 4.2. Meanwhile, we introduce how these problems are solved in current sender-driven approaches and whether these methods are applicable in receiver-driven approach. Then, we present our solutions of these problems in Section 4.3. We validate our solutions by experiments in Section 4.4 and conclude in Section 4.5.

4.2 Problem Statement

4.2.1 Determine Visibility and Visual Contribution of a vertex split

We first define a term *the visibility of a vertex split.* Under a certain viewpoint, if splitting a vertex will help improving the quality of the rendered image, we call this vertex split *visible.* Otherwise, we call it an *invisible* vertex split. In a view-dependent streaming system, it is necessary to determine the visibility of each vertex split so that the sender can avoid wasting bandwidth in sending invisible vertex splits and use the saved bandwidth to send more visible vertex splits.

The visibility of a vertex split is not simply the visibility of the *corresponding vertex* (the vertex to be split by the vertex split), because splitting an invisible vertex may generate visible vertices and improve the quality of the rendered image. To further illustrate this point, we introduce the term *vertex hierarchy.* Vertex splits in a progressive mesh can be organized hierarchically. For example, Hoppe [40] represents parent-child relation among the vertices in

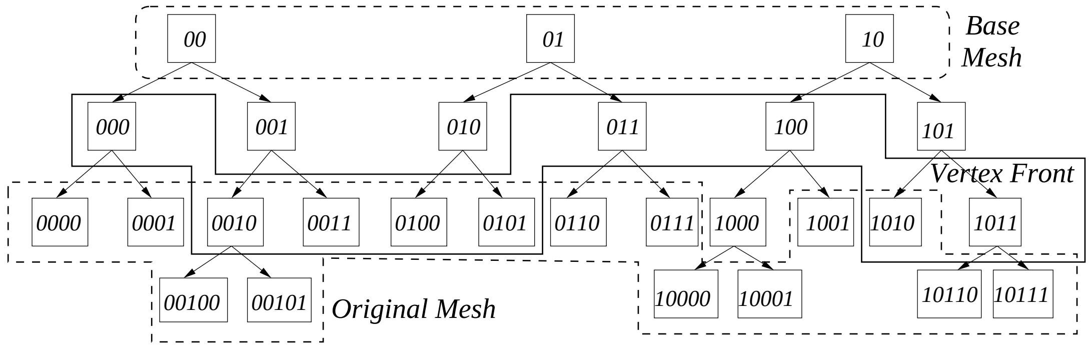


Figure 4.2: Vertex hierarchy and vertex front. A rectangle represents a vertex and the number inside is its identification number, including tree ID and node ID.

a progressive mesh as a forest of binary trees, named *vertex hierarchy*, in which the root nodes are vertices in the base mesh, and the leaf nodes are vertices in the original mesh (see Figure 4.2). A vertex split replaces one vertex (V_s in Figure 4.3) by its two children (V_u and V_t in Figure 4.3). Thus, after applying some vertex splits, the result is a mesh lying between the original mesh and the base mesh. The set of vertices in current mesh is called *vertex front* [40] (see Figure 4.2).

Therefore, an invisible vertex still needs to be split if any of its descendants is visible. To avoid determining the visibility recursively for all the descendants (it is too expensive), a common method is to use a bounding sphere to include a vertex and all its descendants. Then, we can safely ignore a vertex split if its bounding sphere falls outside the view frustum. Similarly, a bounding cone of face normals is used in back face culling [40]. Therefore, some extra information to represent the bounding sphere and bounding cone of face normal is needed. A common practice is to pre-compute the radius of the bounding sphere, the semi-angle of the cone of normals, and other parameters and store them with the vertex together for deciding the screen-space error [40, 48].

These bounding object-based methods, however, are not appropriate in our receiver-driven protocol. If these methods are applied to a receiver driven protocol, the sender has to send the four bounding parameters together with the vertex splits, almost doubling the data size. To avoid increasing the data size, we need a method for the receiver to estimate the visibility and contribution of splitting a vertex without any extra information. We explain our solution to this issue in Section 4.3.

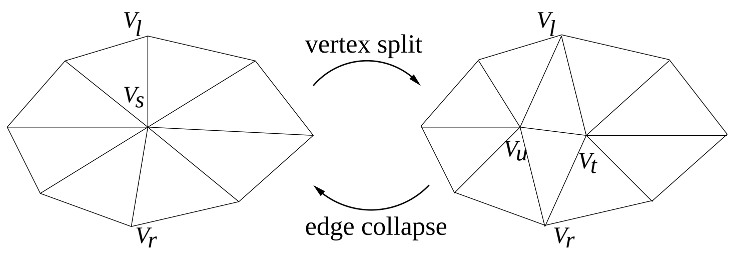


Figure 4.3: A vertex split split V_s to V_t and V_u, with V_l and V_r as two cut neighbors.

4.2.2 Identify Vertices

After deciding the visibility, the sender sends the chosen vertex splits to the receiver. Six parameters are needed in a vertex split of a manifold mesh: the identification number (ID) of the vertex to be split (V_s in Figure 4.3) Ids, the IDs for two cut neighbors (V_l and V_r in Figure 4.3), Id_l and Id_r, and the coordinates x, y, and z of the right child (V_t in Figure 4.3)[1]. In this section, we discuss how to assign the ID to each vertex implicitly.

Many sender-driven implementations use the sequence number of a vertex generated as its ID, then the IDs of two newly generated vertices can be implicitly known by the receiver without extra information. This method, however, forces the sender to be state-ful since the sender has to remember the sending order of vertex splits for each receiver. Therefore, it is not applicable in our receiver-driven protocol, where the sender is stateless and does not remember the sending order.

Kim and Lee [47] proposed a new method, in which every vertex has an ID that is independent of the sending order. The ID of a vertex is a bit string with two parts: tree ID and node ID. Tree ID is the sequence number of the root of this tree in the base mesh, and the node ID represents the path from the root to this vertex in the binary tree. For example, if the tree ID is '01', which is also the ID of the root vertex of this tree, the bit string '010' and '011' are the IDs of the left child and right child of the root vertex respectively. A vertex hierarchy with the assigned IDs is shown in Figure 4.2.

Besides being independent of the sending order, another benefit of this scheme is that the IDs embed the hierarchy. Thus, we can deduce the IDs of the ancestors and the descendants of each vertex. For example, given '1001' as the ID of a vertex, we can deduce that '100' is the ID of its parent, '10010' is the ID of its left child and '10011' is the ID of its right child. This

[1]Here, half collapse is used so V_u remains at the same position of V_s

property frees the sender from sending the IDs of the two newly generated vertices (V_u and V_t in Figure 4.3), as they can be deduced by the receiver.

As a result, with Kim and Lee's method [47], we can identify the vertices implicitly and the identification numbers are independent of the sending order of the vertex splits, which is essential to ensure the sender is stateless.

4.2.3 Compress Vertex Splits Efficiently

As we introduced above, in a sender-driven protocol, a vertex split usually includes six numbers IDs, ID_l, ID_r, x, y, and z. In the receiver-driven protocol, on one hand, the sender needs not send IDs since the vertex splits can be sent according to the requesting order from the receiver. On the other hand, the other five numbers are more difficult to be compressed efficiently.

Encoding ID_l and ID_r. In sender-driven protocol, instead of sending the two cut neighbors, ID_l and ID_r, directly, only the information indicating their position among all the *one ring neighbors* of vertex IDs (the vertices directly connected to vertex IDs) is coded. Since the number of one ring neighbors is much smaller than the total number of vertices, much less bits are needed than coding ID_l and ID_r directly. Moreover, entropy coding can be used to further improve coding efficiency because the vertex ID_l and vertex ID_r are not equally possible in any position.

In the receiver-driven approach, however, the method introduced above cannot be easily applied. Since the receiver may split the progressive mesh in many ways, the set of neighbors of a vertex during the decoding, $\mathcal{N}'$, may not be the set of neighbors during the encoding $\mathcal{N}$. Moreover, a stateless sender does not remember the sending order, so it has no idea what the set is. As a result, even coding on the fly is not possible. In the receiver-driven approach, ID_l and ID_r need to be encoded independent of the decoding order.

Kim et al. [46] proposed an efficient algorithm to encode the IDs of two cut neighbors. With their method, the proper cut neighbors can be selected from the one ring neighbors $\mathcal{N}'$, no matter what $\mathcal{N}'$ is that time. Moreover, even if vertex ID_l or vertex ID_r itself is not in $\mathcal{N}'$ that time, the vertex split still could be applied properly.

Kim et al. [46] also propose an algorithm to efficiently encode the IDs of two cut neighbors. Although their paper focuses on random access of local meshes, we find that this method is useful in progressive streaming as well. We implemented their method and the details are introduced in Section 4.3.

Encoding (x, y, z). As we introduced in Chapter 2, the coordinates of the newly generated vertex, (x, y, z) can be encoded based on the prediction. Usually, the prediction is based on position of some other vertices close to the vertex IDs. This method is not applicable in the receiver-driven approach. Now, because of the random splitting order, it is possible that those vertices used as the base of prediction do not exist yet when vertex IDs is split.

We proposed a simple method to encode the x, y, z based on the position of its parent, the vertex IDs itself. First, the child is usually close to its parent. Second, the parent, vertex IDs here, definitely exist. Otherwise, the vertex split cannot be applied anyway.

We will introduce our implementation of encoding (ID_l, ID_r, x, y, z) in detail in Section 4.3.

4.3 Receiver-Driven Protocol

We now present our proposed receiver-driven protocol for view-dependent progressive mesh streaming. We first introduce the process of transmitting a progressive mesh. Then, we explain how the receiver decides the requesting order. Finally, we explain how we encode the data sent by the sender and the requests sent from the receiver.

4.3.1 Mesh Transmission

A streaming session is initiated when the receiver requests a specific mesh. The sender returns the complete base mesh and other necessary information, such as the Huffman tables used in decoding vertex splits, to the receiver (See Figure 4.1a).

Then, the receiver determines the requesting order of the vertex splits based on the received base mesh, encodes their IDs, and sends them to the sender. On the sender side, the vertex splits are stored in an associative array, which maps the ID to the vertex splits (already in encoded form). After receiving the encoded IDs from the receiver, the sender decodes the IDs and searches for the vertex splits in the associative array with IDs as the key values. The matched vertex splits are sent back to the receiver (See Figure 4.4). The sender does only two things – decode IDs and retrieve the vertex splits, and is therefore stateless.

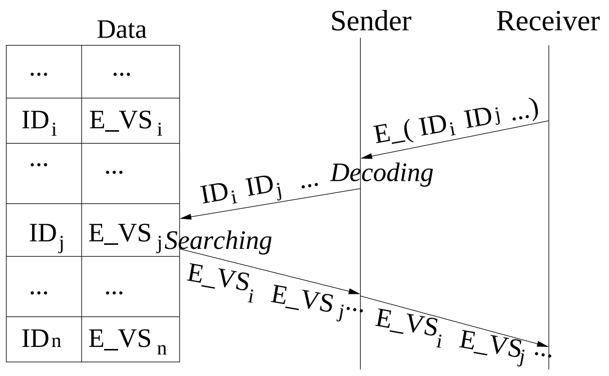


Figure 4.4: The process of the sender in receiver-driven protocol. E represents encoded data, and VS means vertex split.

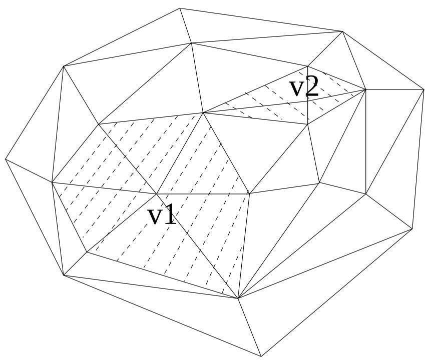


Figure 4.5: Rendered image on the receiver's screen. The shaded are the screen area of vertex V_1 and vertex V_2.

To avoid that the receiver requests a non-existing vertex split (trying to split a leaf vertex in the vertex hierarchy), in the base mesh, an extra bit is attached to every vertex to indicate whether it is a leaf or not. Every vertex splits has two extra bits to indicate whether each of the two newly generated vertices is a leaf.

4.3.2 Determining Visual Importance

We now introduce how the receiver decides the requesting order. We cannot know exactly the contribution of a vertex split in the receiver side because it is requested before it is received. Adding extra information, such as the radius of bounding sphere (a sphere including all the descendant vertices of this vertex), is not appropriate because it will significantly increase the data size. Therefore, we can only estimate the contribution of vertex splits based on the partially received mesh.

We propose to use the screen-space area of all the neighbor faces of a vertex as the metric of its visual importance (see Figure 4.5). The rationale is that if the screen area of a vertex (V_1 in Figure 4.5) is larger, it is likely the quality can be improved by splitting this vertex further. In other words, we use the neighbor faces as an approximation of the projection of bounding sphere used in sender-driven protocol.

We compute the screen-space area with the help of the GPU. First we assign each face in the current mesh a unique color and render them. We can determine the visibility and the screen-space area of each face by counting how many pixels with its color are in the frame buffer, where the rendering result is stored. Then we add the pixel number to the face's three vertices. Thus, the pixel number of a vertex (which is the sum of pixel numbers in its neighbor faces) is proportional to its screen-space area. This process introduce some overhead, but the overhead can be compensated by the time we saved from rendering only visible part of a large mesh later. After the screen-space areas are computed, the receiver sends the requests following the descending order of the screen-space area.

A vertex split will generate two new vertices and their screen-areas need re-computation. Moreover, it also changes the area of some neighbor vertices. Ideally, we should do the screen-area computation after each vertex split and then select the next vertex split. This *vertex-by-vertex* method is not feasible for two reasons. First, doing screen-area computation after each vertex split is too expensive. Second, due to the RTT between the receiver and the sender, the vertex-by-vertex method means to send a request and wait for the vertex splits to arrive before sending next vertex split. As a result, most of the network bandwidth will be wasted in waiting.

To use all the bandwidth available, the receiver should keep selecting vertices to be split and send out the requests. If we only select from the existing vertices, i.e., the vertices in the vertex front, we will face two problems. In the beginning, when the mesh is still coarse, the receiver may have no enough requests to send before the vertex splits already requested arrive after one RTT. Then we cannot fully utilize the network bandwidth when we need it the most. Moreover, even when there are enough vertex splits to request during one RTT, only sending the splits of existing vertices may delay splits of some important children vertices. For example, if vertex v has the highest importance, and if it is split, its two children v_1 and v_2 are the next most important vertices. However, because these two vertices do not exist yet, they will not be selected until the vertex splits of v is received and applied after one RTT.

To address this problem, we also estimate the visual importance of the children vertices when their parent vertex is selected. Whenever a vertex is selected to be split, its two children are added into the candidates. Their virtual importance are both estimated as the half of the selected vertex. For example, if vertex v has importance a_v is selected, its two children v_1 and v_2 are added to the candidates with importance $\frac{a_v}{2}$. Then they will also be considered for the next selection. This estimation introduces some error, and the error could propagate to lower levels. To reduce the error propagation, we periodically do the visibility test again to update the visual importance of each vertex[2]. In Section 4.4, we show that with periodical updating, this method works well. Compared with the ideal case that uses a vertex-by-vertex approach with no network latency, this method has comparable performance.

The effectiveness of the screen-based method is evaluated in Section 4.4, compared with two other methods: One is based on the level of a vertex in the vertex hierarchy and the other is randomly pick a visible vertex to split.

During the streaming, when the receiver is satisfied with the rendering quality, it can either stop requesting or continue requesting the remaining vertex splits for future use based on the prediction of the user's next action. Chapter 5 discusses whether the user actions can be predicted.

If the receiver stops requesting when the visual quality is satisfactory, it may miss some visible vertices. Unlike the sender-driven approach, where the visibility of a vertex and all its descendants can be determined by its bounding sphere, we estimate the visibility of a vertex and its descendants based on the visibility the neighbor faces of this vertex. This estimation may introduce some error (mostly on the silhouette), but fortunately, in most cases this kind of error is small and tolerable (See the experiment results in Section 4.4).

We can reduce the error by splitting the vertices close to the silhouette even when they are invisible. During the process of determining visibility, we can also detect the silhouette. Recall that we assign a unique color to each face. Lets define the set of colors used for a mesh as $\mathcal{C}$. Then, whenever we find a pixel with color in $\mathcal{C}$ is a neighbor (any of the four directions: up,

[2]If the viewpoint changes due to the user's interaction, the visual importance will be re-computed and a new list of vertex splits will be requested.

bottom, left, and right) of a pixel with color not in $\mathcal{C}$, we know that the corresponding face of this pixel is in the silhouette. We can then split the neighbors of this pixel disregarding their visibility. In Section 4.4, we can see that this method helps reduce the error significantly. Moreover, we show that we could assign a higher priority to the silhouette so that the reconstructed image has a more correct shape early. This is useful in some applications where the shape of the mesh is important.

Detecting the silhouette is also useful during progressive streaming. The vertices in the silhouette could be assigned higher weight so that they are split earlier to generate a more accurate profile of the mesh. The weight can be configured by the users according to their preferences. In Section 4.4, we evaluated two screen area based methods, with the weight of silhouette set as 1 and 3, respectively.

4.3.3 Encoding of Vertex Splits and IDs

In this section, we explain how we encode the vertex splits and the IDs of the requested vertex splits. Note that, in our work, we consider only manifold meshes and use half collapse (an edge is always collapsed to one of its two ends) in mesh simplification.

Encoding Vertex Splits To encode a vertex split, we need to encode the IDs of the two cut neighbors and its x, y, and z coordinates. We use Kim et al.'s algorithm [46] to encode the IDs of the cut neighbors (Id_l and Id_r). Since the neighbors of a vertex is a much smaller set than all vertices, a shorter code than the ID is sufficient to differentiate the cut neighbors from other neighbors. Let $\mathcal{N}$ represent the set of the IDs (Here IDs are bit strings beginning with the first bit after leading 1) of the neighbors of V_s, the vertex to be split. Then we encode Id_l and Id_r one by one (if one cut neighbor does not exist, we set its ID as 0) with Algorithm 2. In brief,

Algorithm 2 Encoding the ID of a Cut Neighbor Id. Input: $\mathcal{N}$ and $Id(Id_l$ or $Id_r)$; Output: a bit string *code*.

```
let $i = 1$;
while $\mathcal{N}$ has more than one member do
    delete the IDs in $N$ with $i$th bit (counting from the left) different from the $i$th bit of $Id$;
    if any ID is deleted then
        insert the $i$th bit of $Id$ to the code;
    end if
    $i = i + 1$;
end while
```

each bit in the *code* excludes one or more neighbors from $\mathcal{N}$ until only the cut neighbor with Id is left. In the ideal case, each bit excludes half of the members in $\mathcal{N}$, so the code length is $log_2 d$, where d is the number of neighbors. In the worst case, each bit of the code only excludes one neighbor, so the code length is $d-1$. After obtaining the two codes for Id_l and Id_r, we encode them with the Huffman coding algorithm.

The decoding of these two codes is tricky. Since we enable splitting a mesh in random order, the set of neighbors of a vertex during decoding $\mathcal{N}'$ may not be $\mathcal{N}$, the set during the encoding. If a cut neighbor with ID as Id is not in $\mathcal{N}'$, then either one of its ancestors or some of its descendants are in $\mathcal{N}'$ [46]. In the former case, with the *code*, the ancestor can be found and used as the cut neighbor since its ID is the prefix of Id (recall the property of the unique identification number scheme). In the latter case, with the *code* we find the descendants of the original cut neighbor in $\mathcal{N}'$ since they all have Id as their prefix. Kim et al. [46] proposed a method to find the proper one as the cut neighbor and they show that despite using replacement in vertex splitting, the original mesh can be reconstructed with all the vertex splits are applied. It is also verified by us both in theory and experiments.

To encode the coordinates, instead of encoding x, y, and z directly, we encode dx, dy, and dz with Huffman coding. Here $dx = x - x_0$, $dy = y - y_0$, and $dz = z - z_0$, and x_0, y_0, and z_0 are the coordinates of V_s, the vertex to be split. The rationale to code the differences is that they have less entropy, especially for the later part of vertex splits, which only change the coordinates slightly. It is worth noting that all the encoding processes are done off-line and the encoded vertex splits are stored in the associative array, so the encoding will not increase overhead to the sender.

According to the results of our experiments with the Happy Buddha and Thai Statue, we can quantize dx, dy, and dz to 14 bits. We need 11 bpv for both Id_l and Id_r and 20 bpv for all three of dx, dy, and dz on average. It is worth noting that more bits are needed for dx, dy, and dz for the earlier vertex splits (about 30 to 35 bpv) since their values are larger. The number of bits needed decreases significantly for later part of the vertex splits as dx, dy, and dz decrease. We think that compressing x, y, and z based on better prediction techniques may further increase the efficiency and it will be an interesting topic of future work.

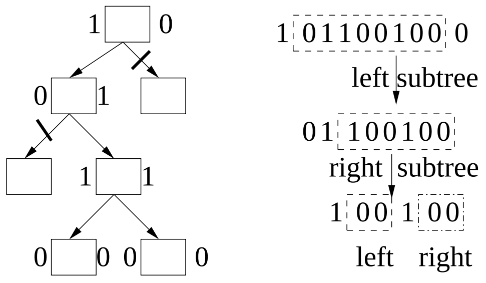


Figure 4.6: The code of ID of the bottom two vertices is 1011001000.

Encoding IDs of vertex splits We now introduce how we encode the vertex split IDs sent from the receiver to the sender. Since we use a 32bit integer to represent an ID, 32 bpv are needed for each ID without compression. It is already larger than the vertex split (31 bpv on average) corresponding to the ID, so compression is desired.

We separate vertex split IDs into blocks, and one block will be compressed as a unit and be sent as a packet. Inside a block, we first filter out all the vertex split IDs whose parent ID is also in this packet. They can be encoded in a special way introduced later. We then focus on those remaining vertex split IDs.

The two parts of an ID, tree ID and node ID, are encoded separately. We use a bit string *code* to store the encoded result. First, we sort the IDs in a packet according to the tree IDs, in increasing order. Then, we store the first tree ID to *code* and store each of the following tree IDs as the difference from the previous tree ID. Since they are sorted, the differences are positive and relatively small numbers.

Algorithm 3 Encoding Vertices in One Tree. Input: IDs of vertices in a tree to be split; Output: a bit string as the *code*.

```
if no vertex needs to be encode in the left subtree then
  append '0' to code;
else
  append '1' to code;
  encode the left subtree;
end if
if no vertex needs to be encode in the right subtree then
  append '0' to code;
else
  append '1' to code;
  encode the right subtree;
end if
```

Next, we encode the node IDs in a tree into a bit string with a recursive algorithm (See Algorithm 3). In brief, we use two bits to represent whether one or more descendants need to be split ('1' for yes and '0' for no) in the left subtree and right subtree respectively. In the example shown in Figure 4.6, for the root vertex, since at least one vertex in the left subtree needs to be split, we append '1' to the code and encode the left subtree. At the root of the left subtree, since no vertex needs to split in its left subtree, we append '0' and check its right subtree. Vertices to be split exist in the right subtree, so we append '1' and encode its right subtree recursively as '100100'. Finally, we return back to the root and append '0' since no node in the right subtree needs to be split. Therefore, the result is '1011001000'.

During decoding, the sender traverses the tree according to the bits of the code. The bit '1' means to decode the subtree and the bit '0' means to stop and return. If a vertex has no descendants that need to be decoded, then this vertex is split. Decoding is done when the procedure returns to the roots.

The advantage of this method is that the code length is variable and the length can be determined without extra flags. The coding efficiency depends on how many vertices need to be split inside a tree. Two bits are assigned to each vertex traversed during the encoding (including the vertices to be split and their ancestors in their path to the roots). Thus, the code efficiency is higher when more vertices in one tree are encoded since the overhead is amortized across the vertices. According to our experimental results, these vertex split IDs can be coded in 18 bpv on average.

At last, for each vertex, we indicate how many levels of descendants are also requested. We use '0' to indicate no descendants, '10' to indicate one level of descendants, '110' to indicate two levels of descendants. In the worst case, when there are no descendants, we add only one bit for each vertex split ID. But once there are some descendants to be split, this method encodes their IDs efficiently (1 bpv for one level, and 0.5 bpv for two levels). We force the descendants at the same level be requested together, and this rule is reasonable. Recall that we assume the importance of the descendants at one level are equal, and is equal to half of the parent's importance.

4.4 Validation

In this section, we introduce the experimental results to validate the screen-area based method.

We test the effectiveness of screen-area based method by checking how fast the quality of rendered image is improved. PSNR values are used to measure the quality of a rendered image, with the rendered image of the original mesh as the reference. The PSNR value tends to increase when more vertices are split, and plotting the PSNR value versus number of vertices split will give us the quality curve. We evaluate the requesting order by examining the quality curve, and sometimes, when PSNR value is not enough to represent the quality of an image, we will show the images directly.

We test our method on various meshes, viewpoints, and initial status so that our result is more representative.

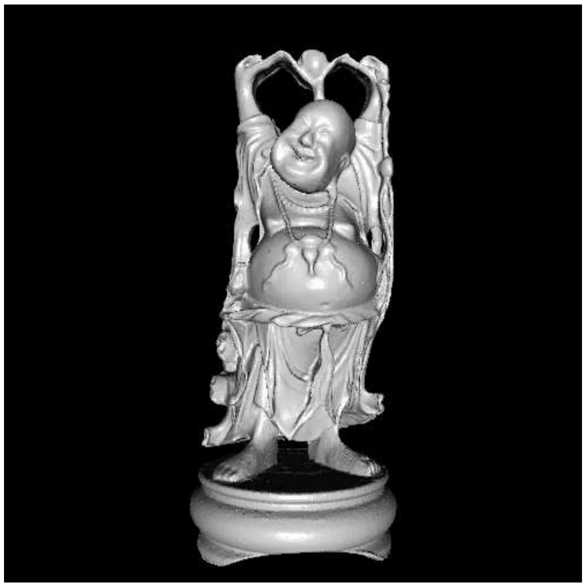
Mesh Two meshes, Happy Buddha (543,652 vertices) and Thai Statue (5,000,000 vertices), both freely available from Stanford 3D Scanning Repository[3], are used in our experiments to show how our method works on meshes with various resolutions. Figure 4.7 shows the rendered images of the original meshes as the reference image and the initial images, which are rendered from the base meshes.

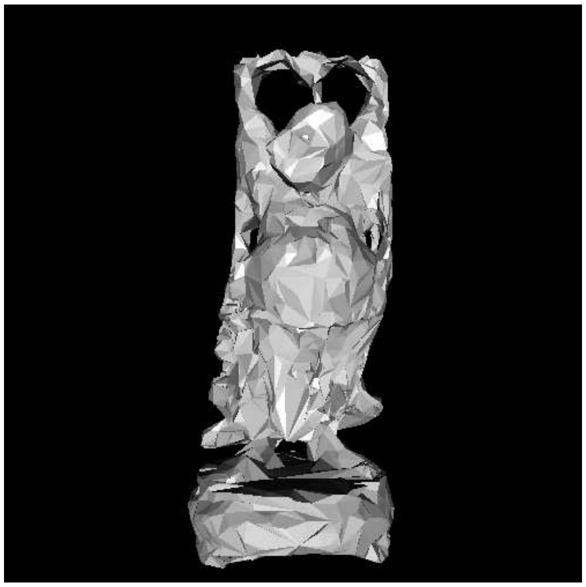
Viewpoint We also include two other viewpoints of Buddha mesh in our experiments (see Figure 4.8). In the first case, the user has zoomed in to see the upper half of the mesh. The second case is a close-up view of the head of the Buddha. The previous viewpoint and these two represent viewpoints with various distance, from far to near.

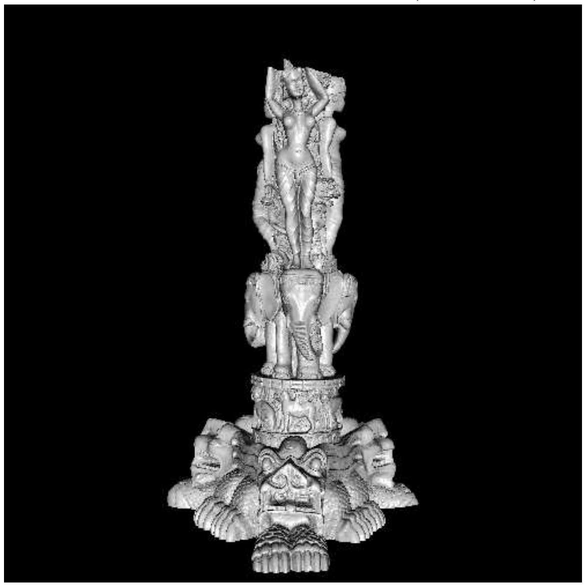
Initial Status When users change the viewpoint, they may see a mesh with uneven quality. The part they have already seen has fine quality, but the newly visible part is still coarse. To address these scenarios, we include the Buddha mesh with different initial status (see Figure 4.9). The first one is generated when the user just zoomed out from the central part of the Buddha mesh, so we can see that the central part of the mesh is already refined but the upper part and the lower part is still coarse. The second one is generated when the user revolved the mesh from left to right, so we can see the right part is refined and the left part is coarse.

Now we begin to introduce the experimental results. In Section 4.3.2 we explained why we estimate the visual importance of the two children vertices when their parent vertex is selected

[3]http://graphics.stanford.edu/data/3Dscanrep/


Happy Buddha Viewpoint 1 (reference)


Happy Buddha Viewpoint 1 (initial)


Thai Statue (reference)

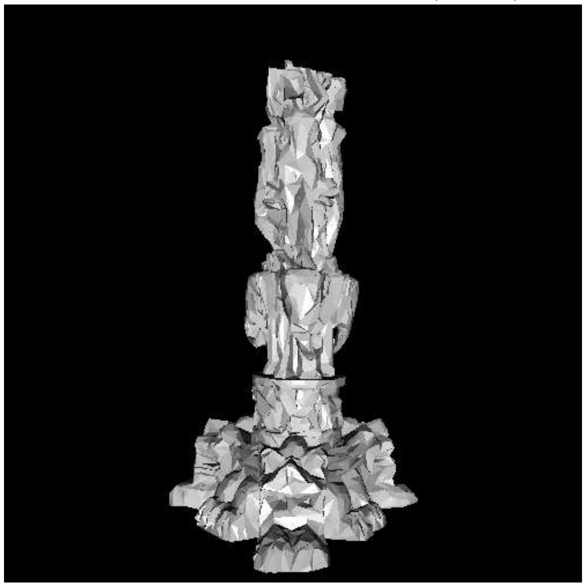
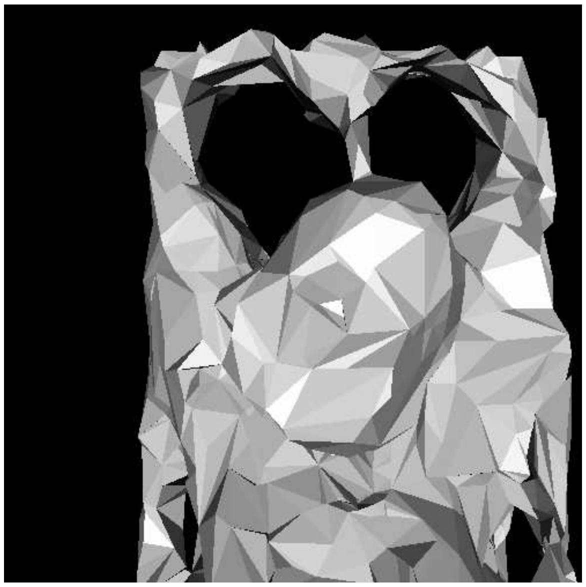
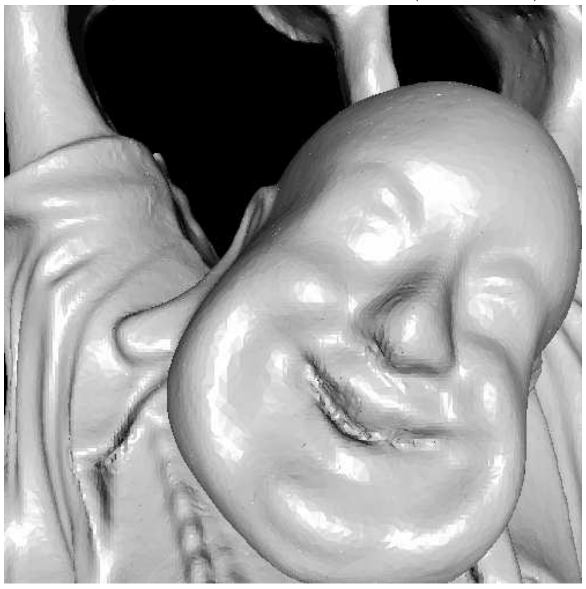

Thai Statue (initial)

Figure 4.7: Meshes we used in the experiments.

Happy Buddha Viewpoint 2 (reference)


Happy Buddha Viewpoint 2 (initial)


Happy Buddha Viewpoint 3 (reference)

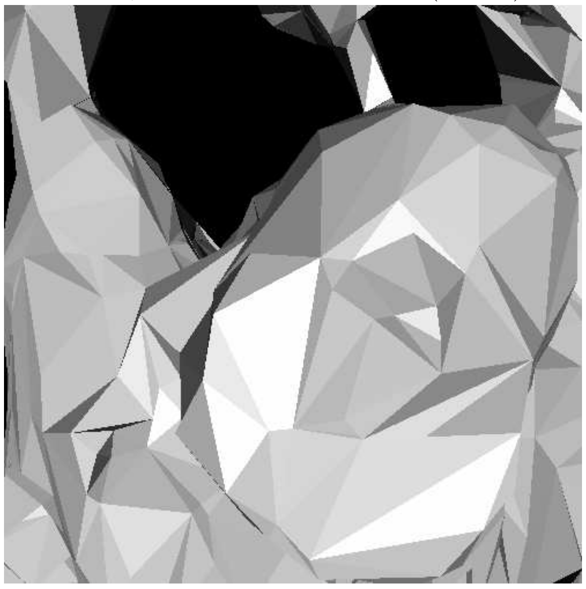
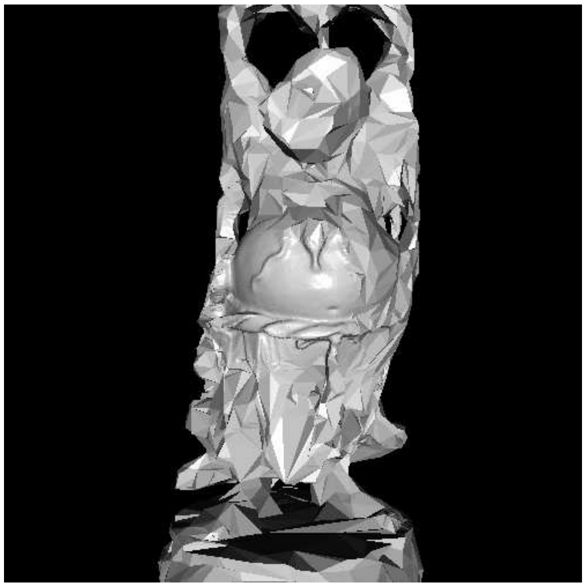
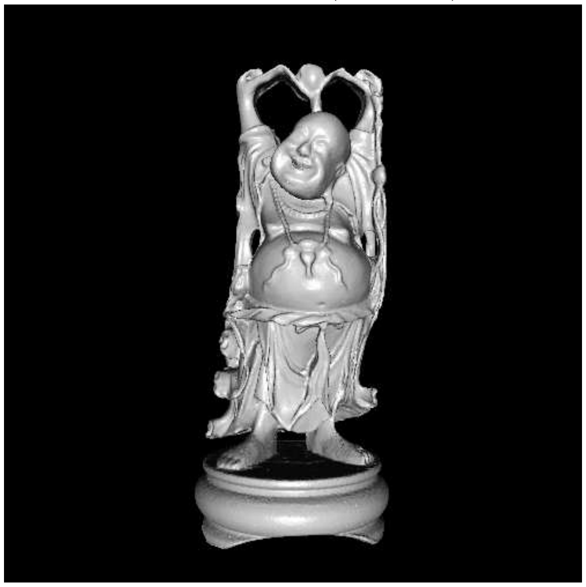

Happy Buddha Viewpoint 3 (initial)

Figure 4.8: Two more viewpoints of Buddha mesh we used.

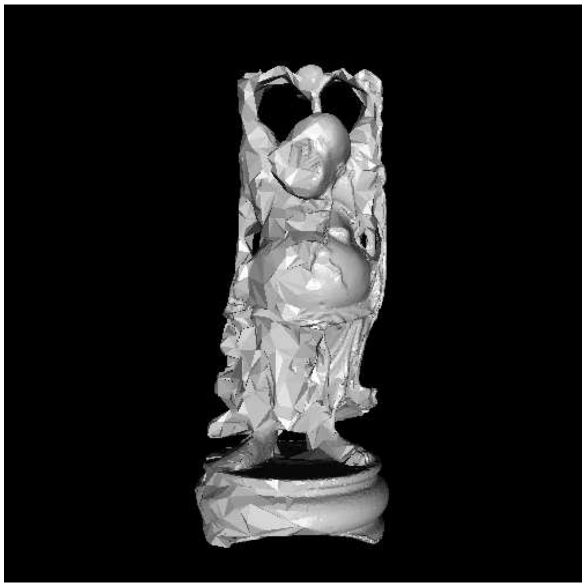
Happy Buddha (reference)



Happy Buddha with initial status 1



Happy Buddha (reference)



Happy Buddha with initial status 2

Figure 4.9: Two more initial status of Buddha mesh we used.

to be split. Here, we show how much it improves the quality. During this evaluation, We assume the RTT is 400ms, the bandwidth is 1Mbps, and the size of a vertex split is 40 bits. As a result, the receiver needs to send 10000 vertex splits to fill the pipeline between receiver and the sender. From Figure 4.10, we see that without considering children vertex splits, during the first several rounds, the receiver cannot find 10000 visible vertices to fill the pipeline, and a considerable part of the bandwidth is wasted in waiting for new vertices to be generated after vertex splits are received from the sender. On the contrary, when we add the children vertices into the consideration, the receiver can always have enough vertices to split, hence the quality curve increases continuously.

Splitting vertices will refine the mesh and change the virtual importance of the vertices. Therefore, we periodically re-check the frame buffer to update the visual importance of each vertex. Next, we show how the updating period affects the quality curve. In Figure 4.11 we show the quality curves for four different updating periods: 1 vertex split, 200 vertex splits, 15000 vertex splits, and 30000 vertex splits. We generate the curve with an updating period of 1 by doing visibility check after every vertex split and assuming vertex splits are available immediately after the request. This ideal case is used as the reference for comparison. This figure shows that when the update period increases, the performance is slightly decreased. The performance is still acceptable with an update period of 30000, which corresponds to 1.2 seconds if the bandwidth is 1 Mbps. Therefore, using GPU to decide the visibility and visual importance can work on devices with limited rendering ability. We just need to set a longer updating period to reduce the overhead.

Next, we compare the screen-area based method with other two methods to justify our proposal. One of the methods compared is the level-based method. We split the visible vertices in the vertex hierarchy in a breadth-first order. The other one is random method, which just randomly picks visible vertices to split. Moreover, we test two screen-area based methods with the weight of the vertices in silhouette as 1 and 3 respectively to show the effect of giving higher priority to the silhouette during transmission. Figure 4.12 shows that two screen-area based methods and the level-based method work much better than the random method. If based on only PSNR value, the level-based method has similar performance in many scenarios and even is slightly better in some cases. The PSNR value, however, sometimes cannot fully

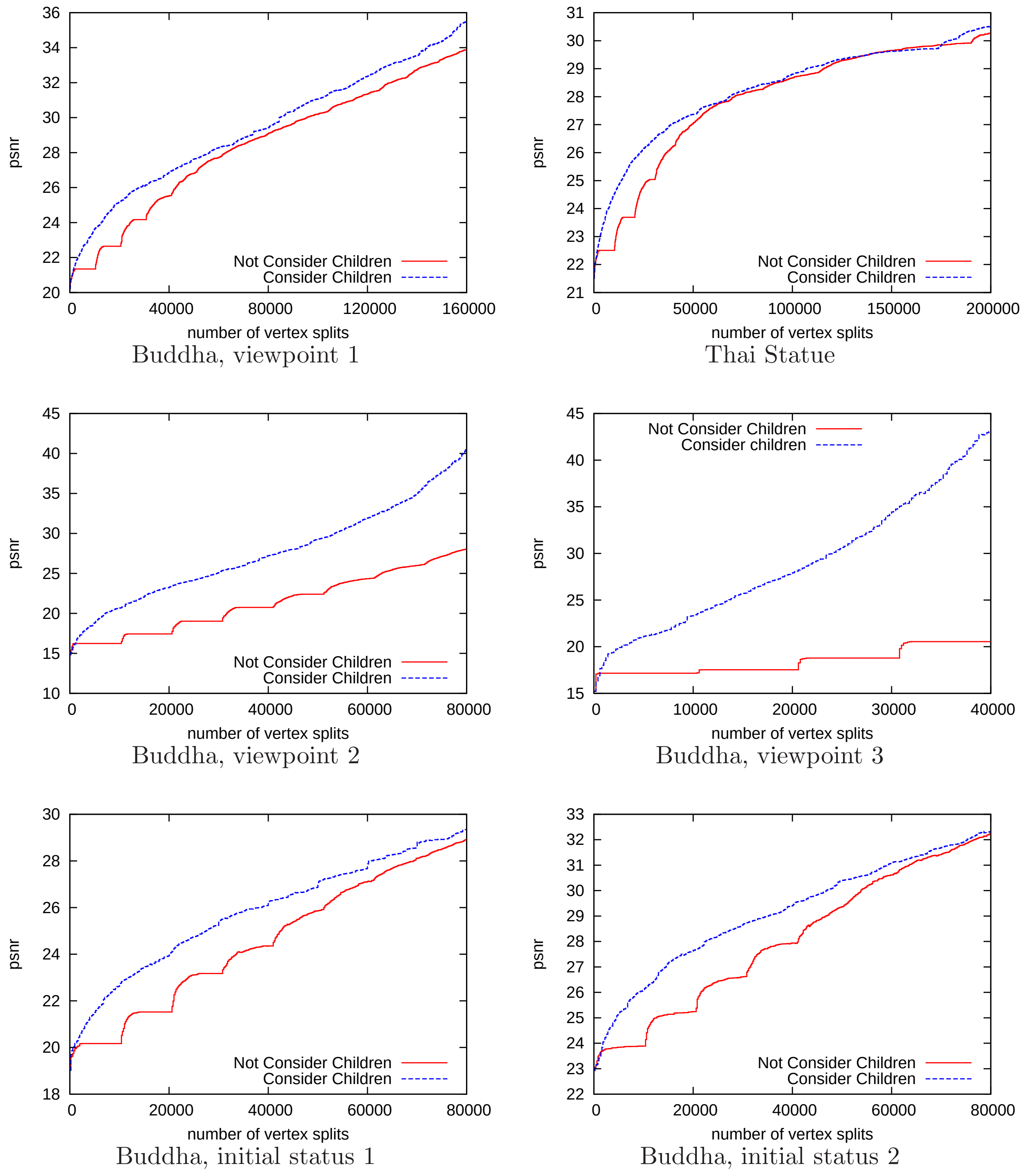


Figure 4.10: The quality curve is improved when children vertices are considered when selecting vertices to split.

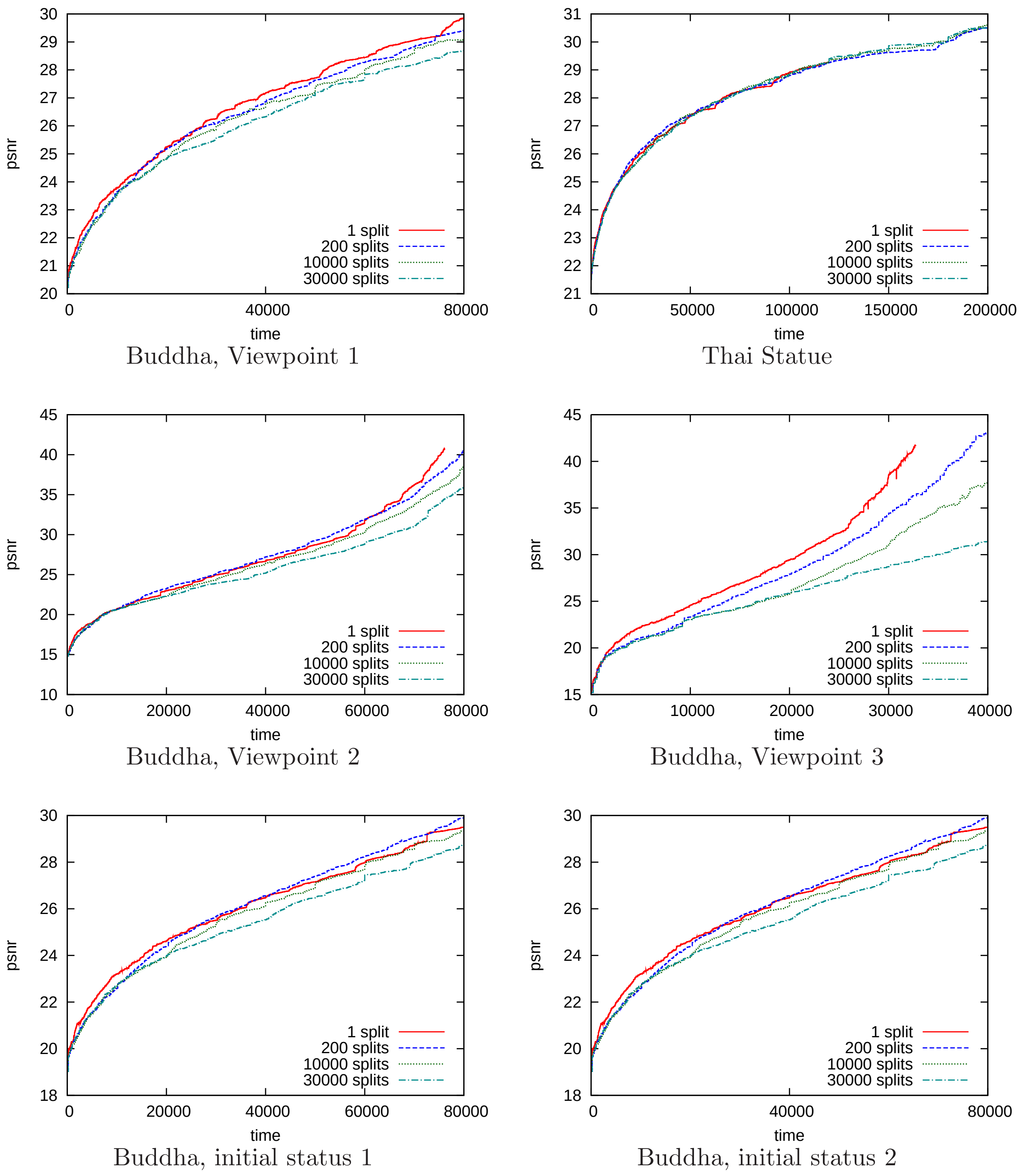
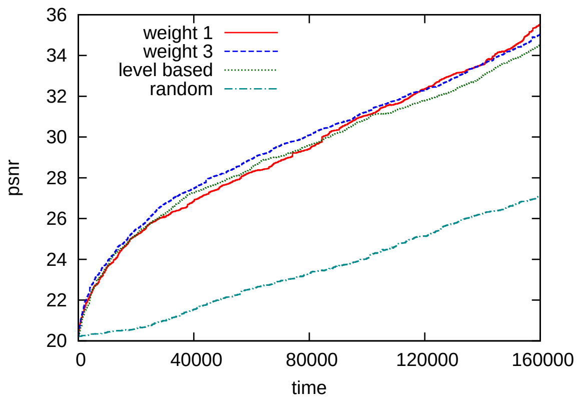
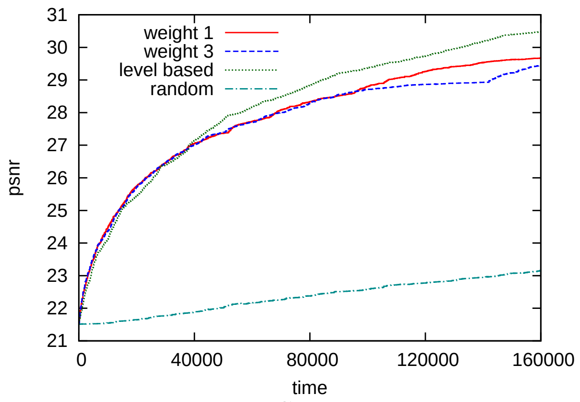


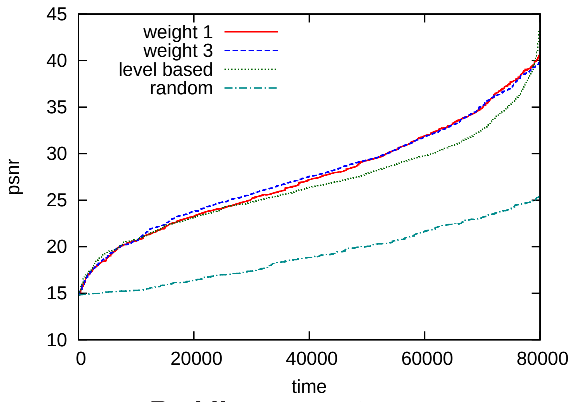
Figure 4.11: Effect of updating period on quality curve.



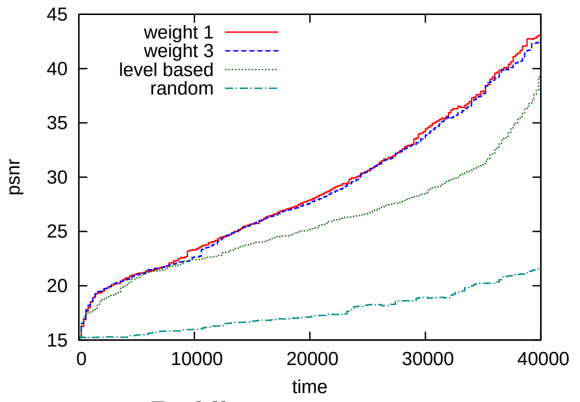
Buddha, viewpoint 1



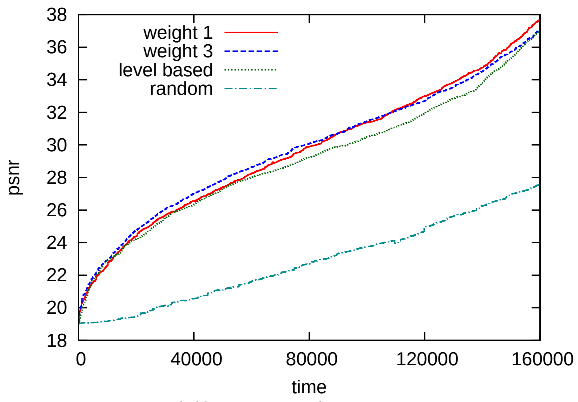
Thai Statue



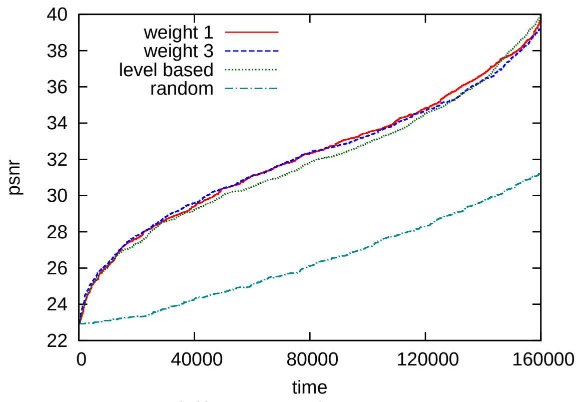
Buddha, viewpoint 2



Buddha, viewpoint 3



Buddha, initial status 1



Buddha, initial status 2

Figure 4.12: Comparing Screen-area method (weight = 1, 3) with level-based method and random method.

represent the visual quality. Figure 4.13 shows the rendered images of Happy Buddha using three methods after 60000 vertex splits. To show the comparison clearer, we enlarge three parts of the images (see 4.14 and show their comparison in Figure 4.15. Although the level-based method generates an image with similar PSNR value, we can see that the quality of the face and base is considerably worse. The silhouette of the level-based method, however, is slightly better than the screen-area based method with weight 1. This is caused by our screen-based method, which tends to under-estimate the importance of the parallel (or near to parallel) faces. We can see that with weight as 3, the rendered image has a more accurate silhouette and at the same time has good quality.

When there are many details on the screen and the user views it from a relative far distance, such as the Thai mesh in our case, the difference between the level-based method and screen-based method becomes hardly noticeable. Although the PSNR value of level-based method after splitting 60000 vertices is slightly higher, we cannot clearly tell which one is better in Figure 4.16, where the rendered images are shown. To compare the quality of these images is not so important, because most likely, the quality for the current viewpoint is good enough and users have already zoomed in to check the details they are interested in. We have not included the closer viewpoints of Thai Statue here, because once zoomed in, the visible part of the Thai Statue is no longer more complex than Buddha mesh. In other words, the results of Thai statue with a near viewpoint are similar to the results of Happy Buddha with a far viewpoint.

Finally, we check the accuracy of the visibility test. As we explained in Section 4.3.2, if the receiver stops requesting vertex splits after all the visible splits are received, some potentially visible vertices may not be generated. We use two metrics to compare the rendered image and the reference image (the rendered image of the original mesh) when all visible vertices are split. One is to find how many pixels are different and the other is to compute the PSNR value. Table 4.1 shows that the error is negligible. Moreover, if we force to split all the neighbors of vertices in the silhouette (as introduced in Section 4.3.2), we can further reduce the error and have a more accurate silhouette.

In summary, our screen-based method improves the quality of rendered image significantly quicker than the random method. Compared with the level-based method, the screen-based method usually generates better rendered images in perception, especially when the viewpoint

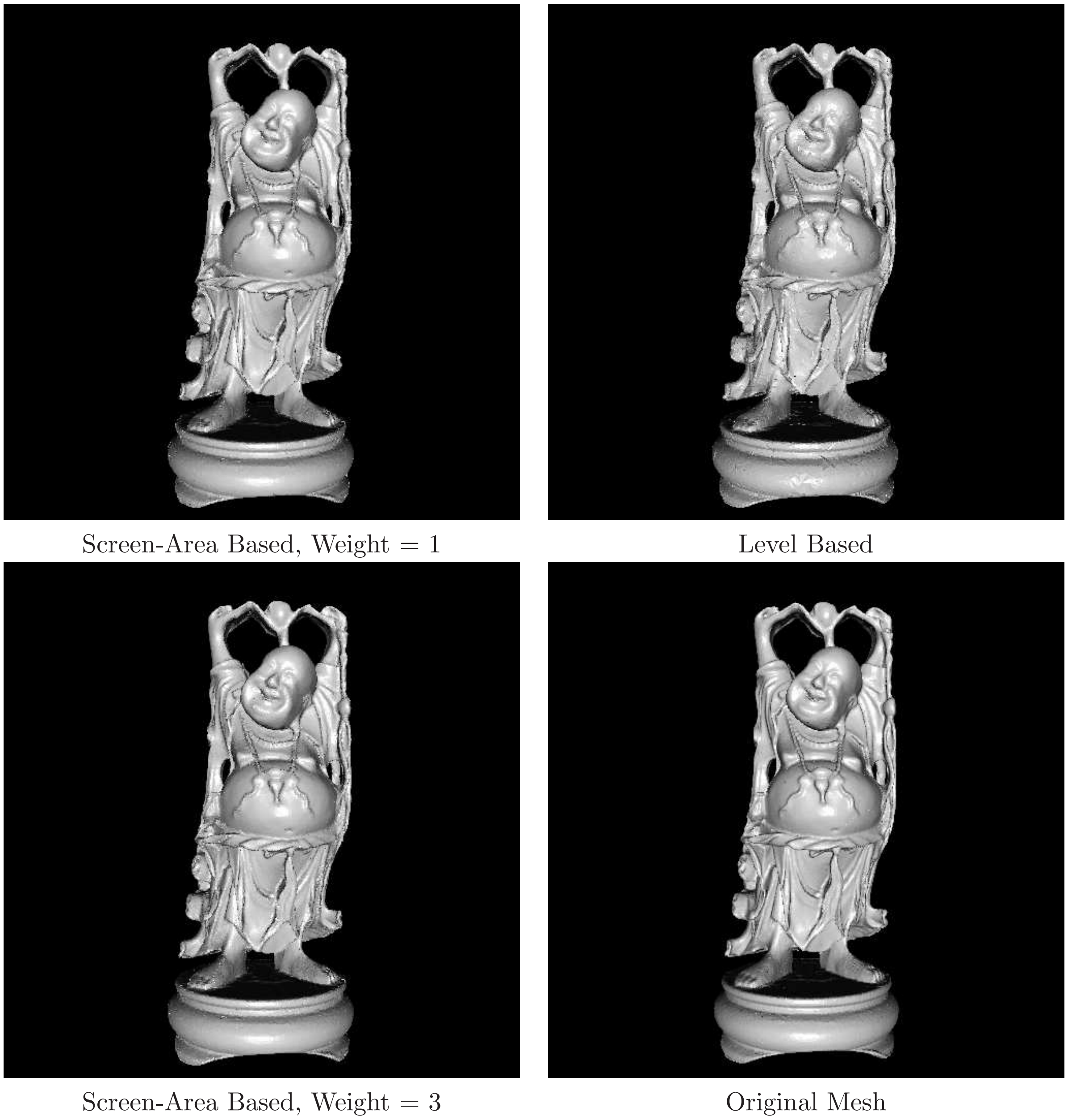


Figure 4.13: Comparing Screen-area method (weight = 1, 3) with level-based method. Happy Buddha with 60000 vertices are split.

Scenario	Without force split		With force split	
	Number of Error Pixels	PSNR	Number of Error Pixels	PSNR
Buddha viewpoint 1	202 (0.081%)	40.8971	112 (0.045%)	43.514
Buddha viewpoint 2	118 (0.047%)	40.8434	44 (0.018%)	44.2068
Buddha viewpoint 3	37 (0.015%)	48.234	22 (0.009%)	49.5148
Thai Statue	372 (0.149%)	36.3953	273 (0.109%)	37.3371

Table 4.1: The error in visibility determination (before and after force splitting silhouette). The resolution is 500×500, so the total number of pixels is 250000.

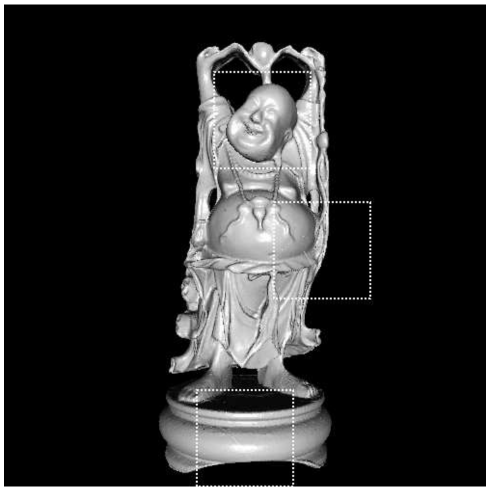


Figure 4.14: Three parts of the Buddha: head, base, and edge.

is close to the mesh. Moreover, by treating the silhouette specially, we can have a better profile of the mesh both in the intermediate state and the final state (after all visible vertices are split).

4.5 Conclusion

In traditional view-dependent systems, senders decide which refinements to send, so they are state-ful and need expensive computation. Besides the CPU time, the output bandwidth also makes the sender not scalable to many receivers. Cache proxy and peer-to-peer system, the two common way to increase the scalability without deploying more servers, cannot be applied easily due to the state-ful design.

We propose a receiver-driven protocol to address the above weakness. Our main finding is that receivers can estimate the visual importance of a vertex split, even before it is received, based on the current partially received mesh. Moreover, we can exploit the GPU in determining the visibility and visual importance of vertices by analyzing the content in the frame buffer. In addition, we could easily detect the silhouette and hence can assign it higher priority to ensure the rendered image has an accurate shape. We have done extensive experiments to validate the screen-area based method.

Now the sender is stateless in our implementation, so the receiver can easily switch from different senders during transmission or even simultaneously download from multiple senders. Therefore, we can easily apply the cache proxy or peer-to-peer techniques to solve the scalability problem.

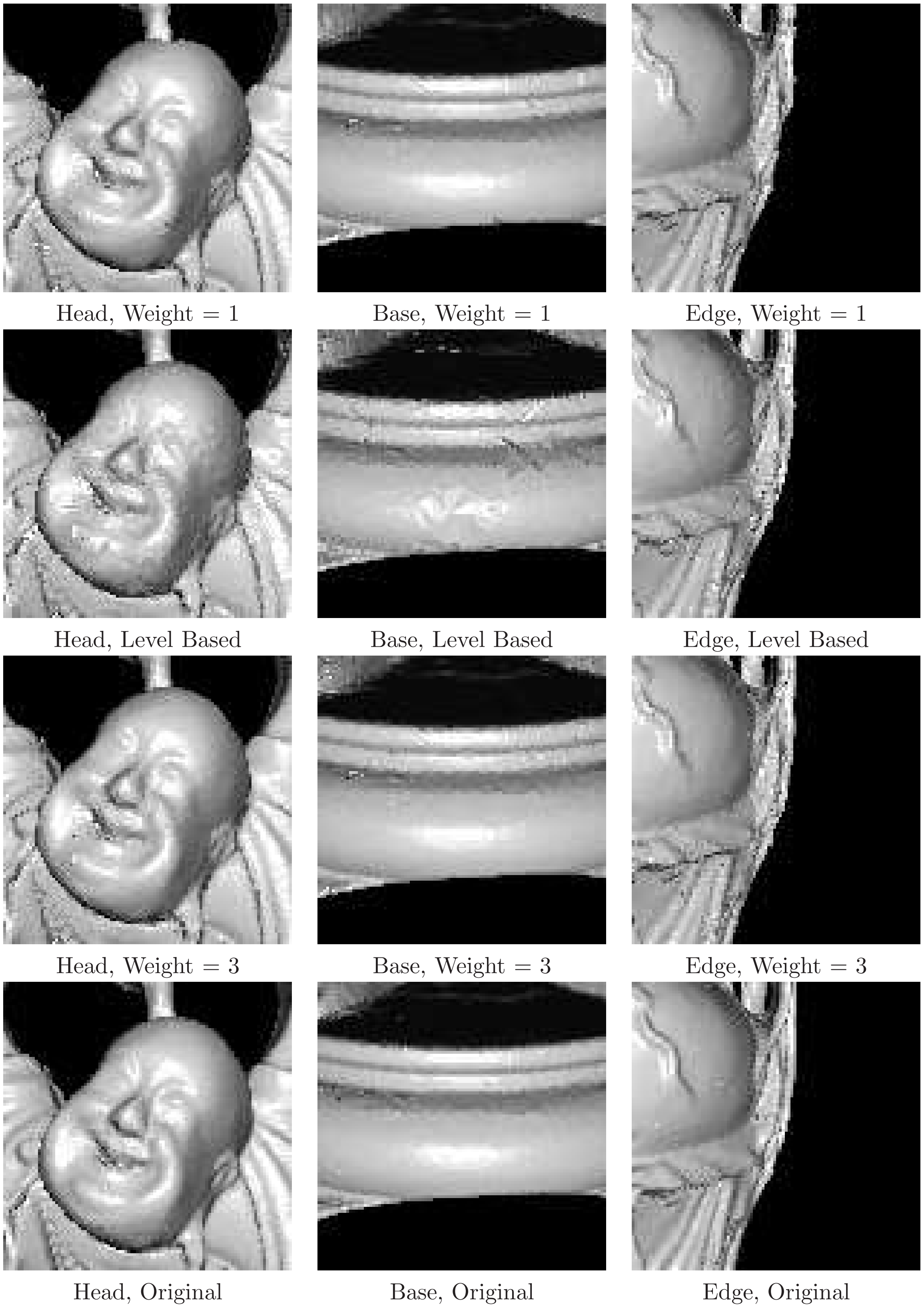
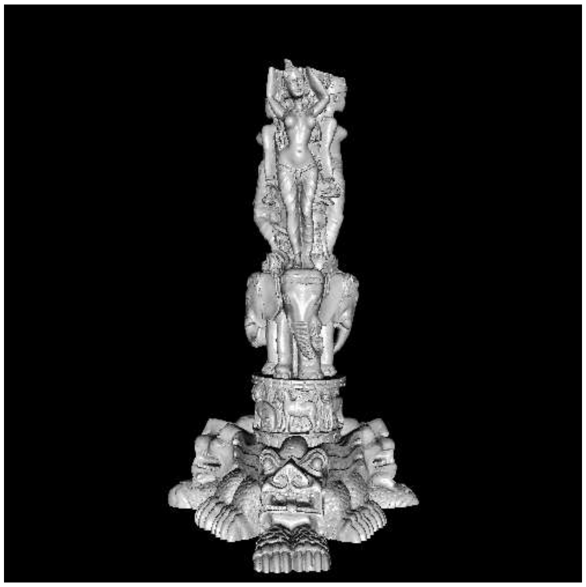


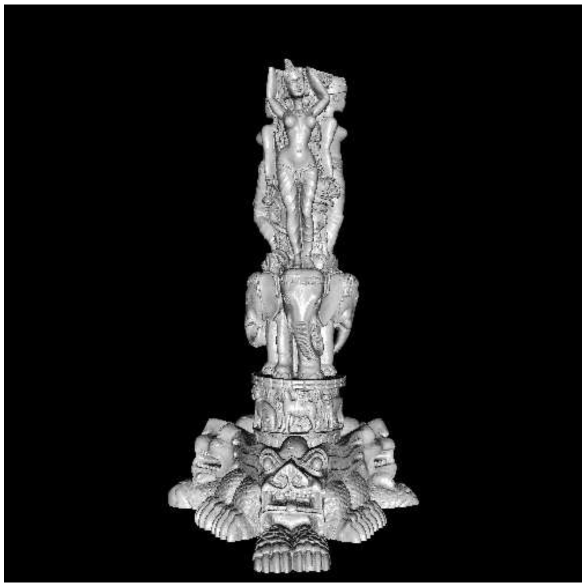
Figure 4.15: The enlarged images of the three parts of the Buddha Mesh

Screen-Area Based, Weight = 1

Level Based



Screen-Area Based, Weight = 3



Original Mesh

Figure 4.16: Comparing Screen-area method (weight = 1, 3) with level-based method. Thai Statue with 60000 vertices are split.

Chapter 5

Characterizing User Interaction with Progressively Transmitted Meshes

5.1 Introduction

In a view-dependent progressive mesh streaming system, which set of the data is streamed and in what order it is streamed both depend on how the user interacts with the progressively streamed mesh. Hence, user interaction needs to be considered when testing a mesh streaming system. The ideal way is to record interaction of a large number of users with a variety of meshes under multiple usage scenarios. This method, however, is expensive and not always feasible. If we could characterize user interaction from a small number of real user traces (the records of the interaction of users), we could create a large number of synthetic traces, which simulate users' actions, to be used in measuring the performance of our system. This method is much cheaper than collecting a large number of real traces from users.

Therefore, we conducted an experiment with 37 users interacting with 9 meshes. We logged the user's actions while they interact and view the meshes in a mock online shop. Then we did some preliminary analysis on the user traces we logged. Based on this analysis, we developed a simple model, which can be used to generate synthetic traces for evaluating our peer-to-peer view-dependent mesh streaming system introduced in the next chapter.

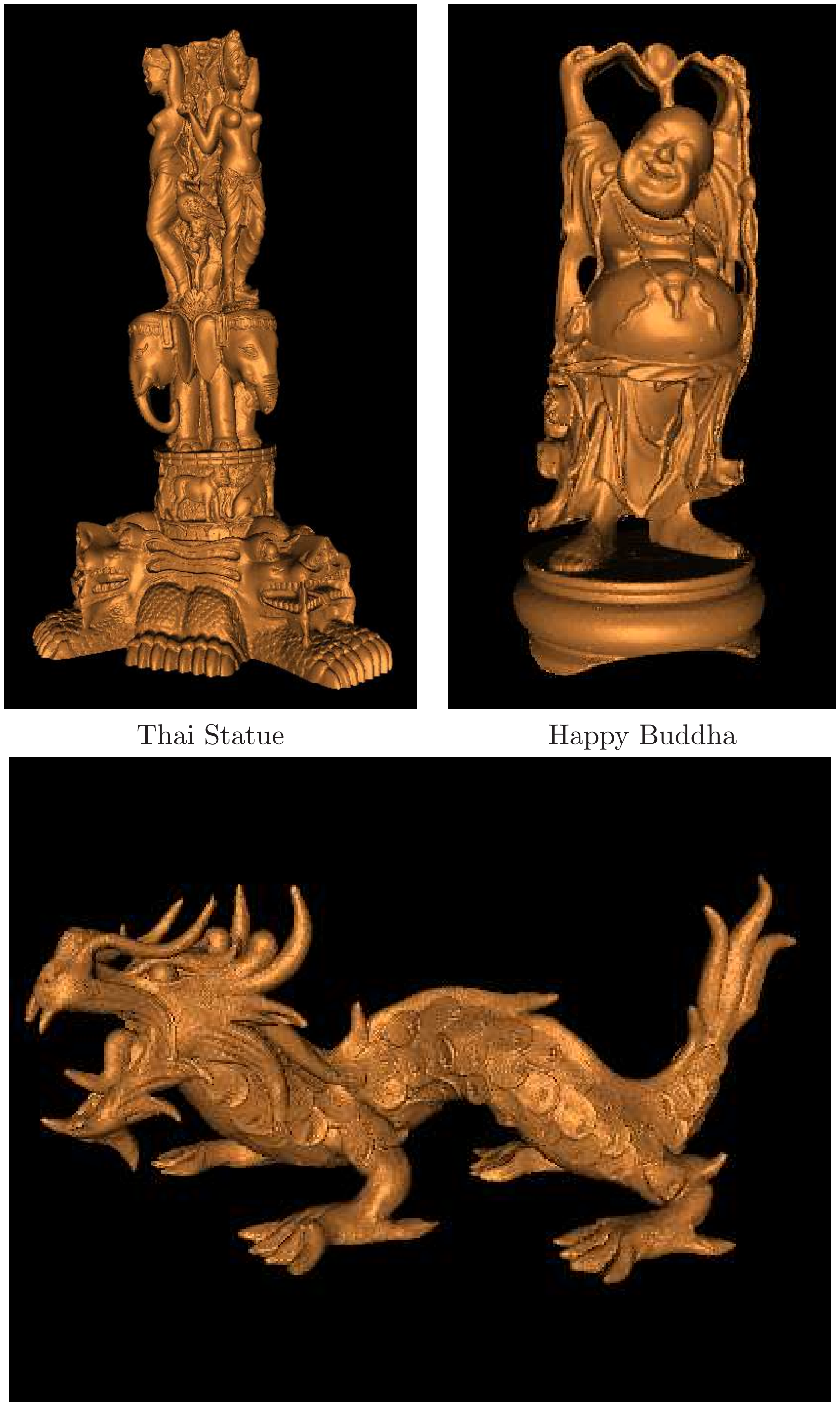


Figure 5.1: Meshes used in experiment. Thai Statue (5 million vertices), Dragon (3.6 million vertices), and Happy Buddha (0.5 million vertices).

We first introduce how we did the experiment and recorded the data. Then we show some interesting findings from our preliminary analysis. Finally, we introduce how we generate synthetic traces and compare them with the real traces we collected.

5.2 Experiment Setup

Meshes

Three 3D meshes are chosen from the freely available Stanford 3D Scanning Repository: *Happy Buddha*, *Dragon*, and *Thai Statue*. These meshes vary in complexity (amount of vertices), orientation, and symmetry in space from default viewing direction. *Happy Buddha* is the simplest, is vertically oriented, and has a default viewing direction orthogonal from the face of the Buddha. From that direction, the mesh is asymmetric between front and back. The geometric shape of *Happy Buddha* is somewhat representative of all human-like statues. *Dragon* is more complex and is horizontally oriented. The default viewing point is from one side of the body. Unlike *Happy Buddha*, it is front-back symmetric relative to the default viewing direction. The geometric shape of *Dragon* is somewhat representative of most mammals. *Thai Statue* is the most complex and is actually a compound mesh composed of three identical sides, each with three different objects: a Goddess, an elephant, and a dragon, stacking vertically from top to bottom. These three sides connect to form a triangular cylinder. There are three possible default viewing directions, one each from three corners of the triangular mesh. The *Thai Statue* is included as an example of complex compound mesh.

We replicate each mesh above twice, generating nine meshes in total. We added one localized visual defect, by denting a small region, to each replicated mesh without changing its number of vertices or faces. The reason for adding the defects will be explained later. The location and nature of the defects vary between the meshes. As *Happy Buddha* is relatively simple and has a smooth surface, its defect is more obvious than *Dragon* and *Thai Statue*. Defects for the later two meshes, due to the irregular surfaces, are hard to find unless the user zooms in considerably.

The meshes are encoded progressively and streamed over a simulated network of 320Kbps and 400ms round trip time.

Participants

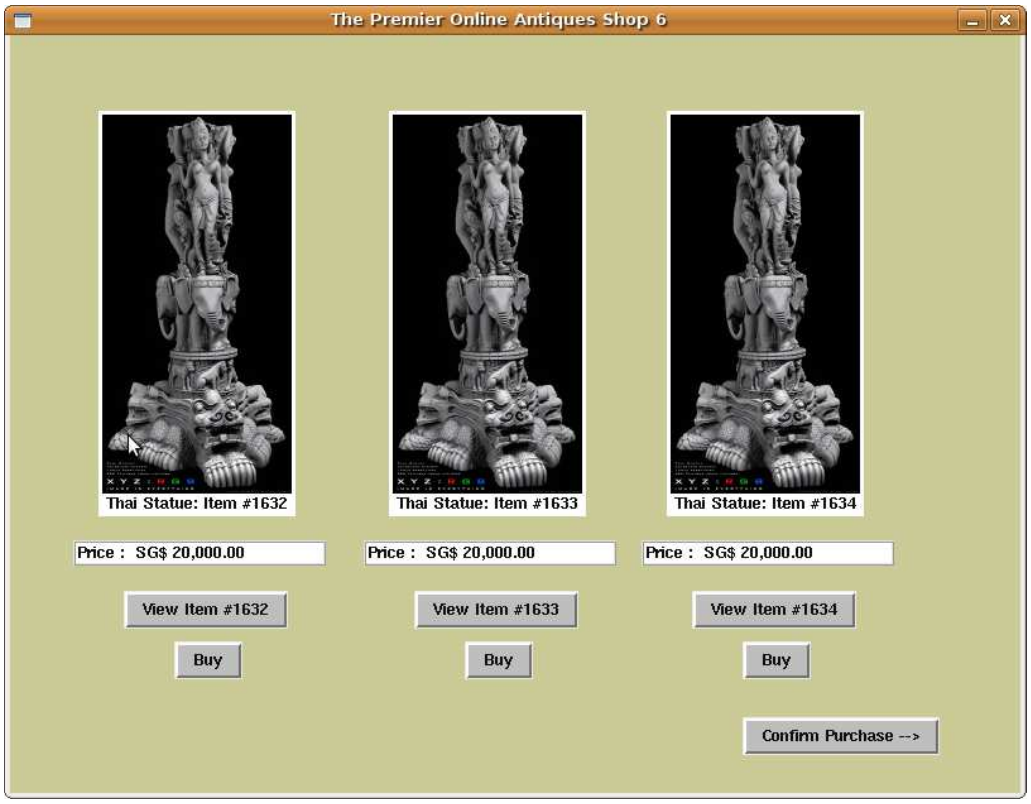


Figure 5.2: The user interface we used to mimic an online shop.

A total of 37 (25 male and 12 female) participants, aged 19 to 36 (mean 23), mostly from the university community participated in the experiment. None had any visual handicaps.

Design and Procedure

Our experiment mimics the following general real world scenario. Customers are shopping in an online antique store. Each product in the store has a number of items, and each item is represented by a 3D mesh closely resembling the corresponding real world item. These items vary in quality, and some have visible defects. Before purchasing, customers will carefully examine the available items for a product in order to pick the best item available for that product category.

We designed our experiment using a simple case of the above scenario. Our online store has three different products, corresponding to the three different meshes mentioned earlier. Each product has three available items in varying quality. Two of them have visual defects (note: defects are created without changing any mesh characteristics). Due to the different complexity of each mesh, the defects are easier to find in the simpler meshes (*Happy Buddha*) compared to the more complex ones (*Thai Statue* and *Dragon*).

The participants were first instructed about the keyboard commands to view and interact with the 3D meshes, and given brief practice of these commands on a simple mesh before starting the experiment. The participants were presented with a user interface mimicking an online catalog with three products (see Figure 5.2). For each product, images of three items (i.e., three versions, one original and two with defects) are shown on the screen. The order of the products is randomized to avoid order effects. The participants' job is to pick the best item among the three. Each item can be viewed in any order and if desired, multiple times. When a participant selects an item, a new viewing window (width of 14cm and height of 15cm, with a resolution of 500×500 pixels) opens, and the 3D mesh corresponding to that item is progressively streamed and rendered in the window. The participants can interact freely with the mesh until they close the window. They would mark the item to be purchased after viewing all three items and move on to the next product. Each participant must go through all three products to complete the experiment.

User Behavior Logs

During the experiment, users' key press actions and viewpoints were logged for off-line analysis. The representation of the viewpoints and actions are introduced in detail as follows.

A viewpoint indicates from where and to which direction users view the mesh. Users can change their viewpoints during the session. Changing the position of the viewpoint is equivalent to changing the position of the mesh. When the application starts, the user begins with the default viewpoint, where the user looks at the center of the mesh (more precisely, the center of the bounding box of the mesh) from a distance far enough to see the whole mesh.

We use a 6-tuple, $(x, y, z, \theta_x, \theta_y, \theta_z)$ to indicate the relative position of the mesh, when the mesh is translated x units right, y units up, and z units closer to the user (i.e. out of the screen), and rotated θ_x degrees around x axis, θ_y degrees around y axis, and θ_z degrees around z axis. The positive directions of translation and rotation are shown in the Figure 5.3. The default position is $(0, 0, 0, 0, 0, 0)$. The position of x, y, and z changes in unit equivalent to one tenth of the bounding length, which is the edge length of the bounding box of the mesh. The angles of θ_x, θ_y, and θ_z change in units of 10°, and their range is $[0, 1, \cdots, 35]$.

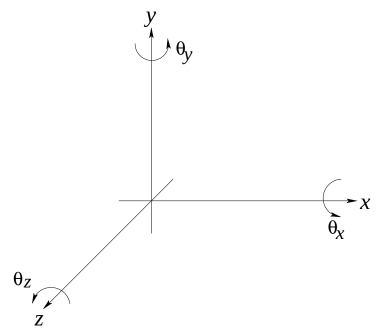


Figure 5.3: The positive direction of translation and rotation.

Name	Abbreviation	Result	key/key combination
move right	MR	$x+1$	ALT + $\rightarrow$
move left	ML	$x-1$	ALT + $\leftarrow$
move up	MU	$y+1$	ALT + $\uparrow$
move down	MD	$y-1$	ALT + $\downarrow$
zoom in	ZI	$z+1$	$\uparrow$
zoom out	ZO	$z-1$	$\downarrow$
tilt backward	TB	$\theta_x+10°$	CTRL + $\downarrow$
tilt forward	TF	$\theta_x-10°$	CTRL + $\uparrow$
revolve anticlockwise	REAN	$\theta_y+10°$	$\rightarrow$
revolve clockwise	REC	$\theta_y-10°$	$\leftarrow$
rotate anticlockwise	ROAN	$\theta_z+10°$	CTRL + $\leftarrow$
rotate clockwise	ROC	$\theta_z-10°$	CTRL + $\rightarrow$
quit	Q	quit	Q

Table 5.1: Introduction of the actions.

To facilitate the analysis, we ensure the six values are all integral. Therefore, an action will change one of the six values by one unit. Table 5.1 lists out the name, the abbreviation, the result, and the corresponding key (or key combination) of each action.[1]

As a result, a user session can be seen as a random process with viewpoints as the states. Each action transits the system from one state to next state, until the QUIT action terminates the session (see Figure 5.4 as an example). During the session, we record all the user actions and the time they happened. We call these records *user trace*. Moreover, $(x, y, z, \theta_x, \theta_y, \theta_z)$ can be seen as a point in a 6-dimensional space, and the six axes in this space are X, Y, Z, AX, AY, AZ. In this sense, a session is a random walk in the 6-dimensional space.

[1]There is a small inconsistency in our design. We use $\uparrow$ to move the mesh closer and $\downarrow$ to move the mesh further. It is more like moving viewpoint than moving the mesh. It is the flaw in our design, but fortunately the users were used to it quickly during the experiments.

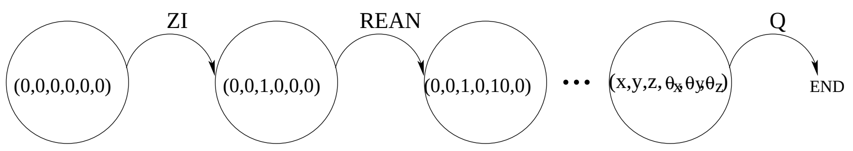


Figure 5.4: An example of a user session

5.3 Predictability and Locality

The original purpose of our analysis on the traces we collected is to find some patterns to help us generate the synthetic traces. Some preliminary findings we obtained, however, are interesting themselves. We found that in the traces we collected, the user actions are somewhat predictable. It indicates the possibility of applying pre-fetching in our mesh streaming system. Moreover, we found that locality exists in the traces we collected and it means caching can be useful in our scenario. In this section, we introduce these two potentially useful by-products of our experiment in detail.

5.3.1 Predictability

It is interesting to see if user actions are predictable. If so, pre-fetching can be used to reduce response time when users change their viewpoints. For example, when the connection between the receiver and the sender has limited bandwidth or long delay, the requests for vertex splits of the new viewpoint can be sent based on the prediction before the user changes the viewpoint. Another example is that when the client has limited rendering power, the next viewpoint can be rendered in advance based on the prediction to reduce the response time.

The next action of a user, A_n, is a random variable with 14 possible values (see Table 5.1). A naive way to predict the value of A_n is based on the unconditional distribution of A_n, which can be estimated as the frequency of each value in the traces we collected. Then, A_n can be predicted as the action with highest probability. We note this action as a_{max} and its probability as p_{max}. Figure 5.5 shows the frequency of occurrences of 12 different actions in the traces we collected, displayed as a bubble chart. The bubble size indicates the frequency of an action occurring for a mesh. We can see that the most frequently used actions are two revolving (rotating around y-axis) actions. Table 5.2 shows the a_{max} and p_{max} for all 9 meshes. As a result, without any extra information, we can always predict that the next action is a_{max}, and the accuracy is

Mesh	a_{max}	p_{max}
Dragon	REAN	0.2517
Dragon1	REC	0.2237
Dragon2	REAN	0.2180
Thai	REAN	0.3292
Thai1	REAN	0.3323
Thai2	REC	0.3375
Buddha	REAN	0.3584
Buddha1	REAN	0.2707
Buddha2	REC	0.3438

Table 5.2: The action with highest unconditional probability and its probability.

p_{max}. The accuracy of this prediction, however, is low. The best prediction we can make is for Buddha, which has the accuracy as 35.84%. The prediction could be more accurate with more information, such as the previous action of the user and the current viewpoint of the user.

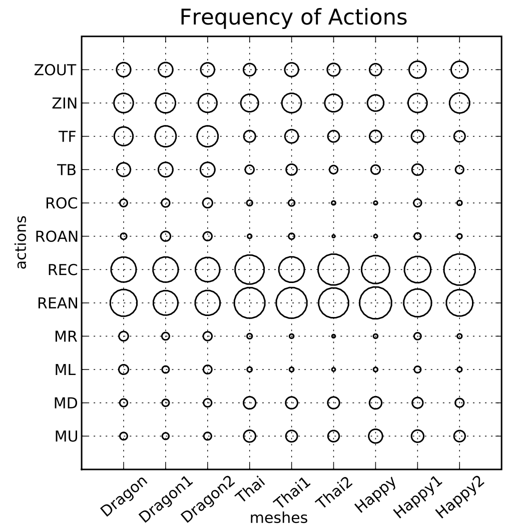
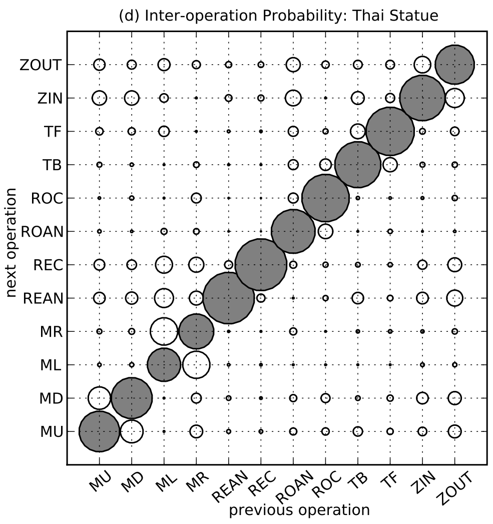
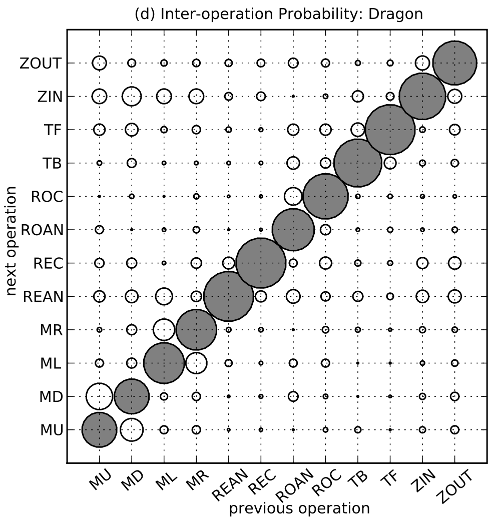


Figure 5.5: The frequency of 12 actions in the traces we collected.

Considering Previous Action From the traces, we found that the value of A_n highly correlates with the value of A_p, the previous action taken by the user. Figure 5.6 shows the conditional probability of taking the next action given the previous action for the three meshes. The shaded diagonal shows that the same action has the significantly larger probability (e.g., more than 0.93 for revolving) of being taken next.

According to this observation, we categorized the actions into two types: *repeat*, *non-repeat* and use P_{repeat} to represent the probability to repeat the previous action. Then we can predict the next action by choosing "repeat" unconditionally. The P_{repeat} for three meshes can be seen





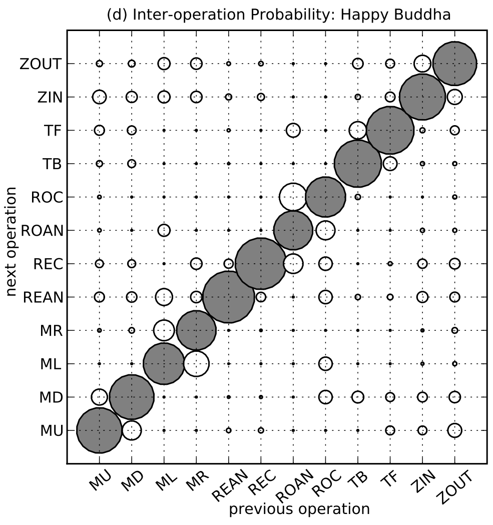


Figure 5.6: The conditional probability of next action if the previous action is given.

in Figure 5.7. We can see that always choosing "repeat" as the prediction has high accuracy (larger than 80% for all meshes).

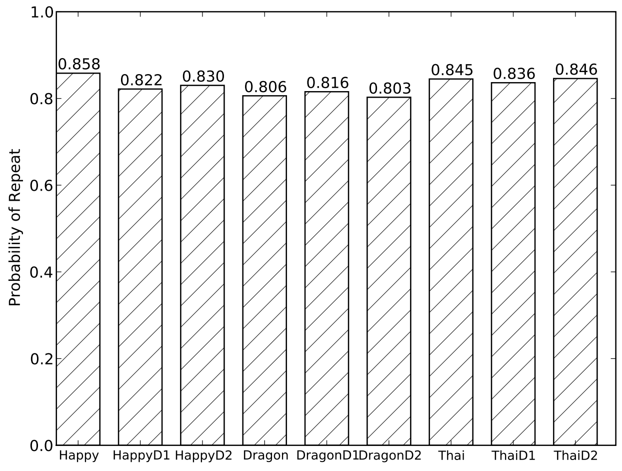
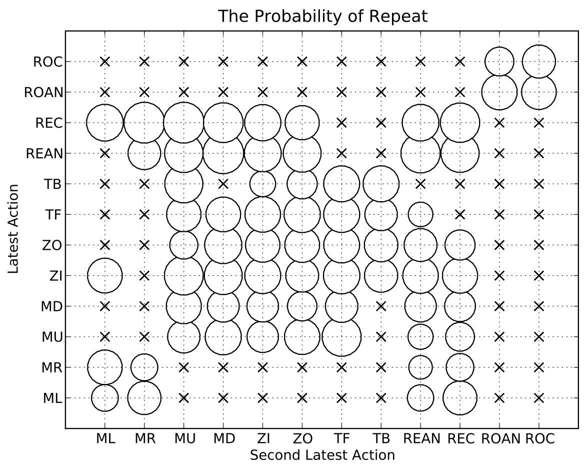


Figure 5.7: The probability of repeating the previous action for 9 meshes.

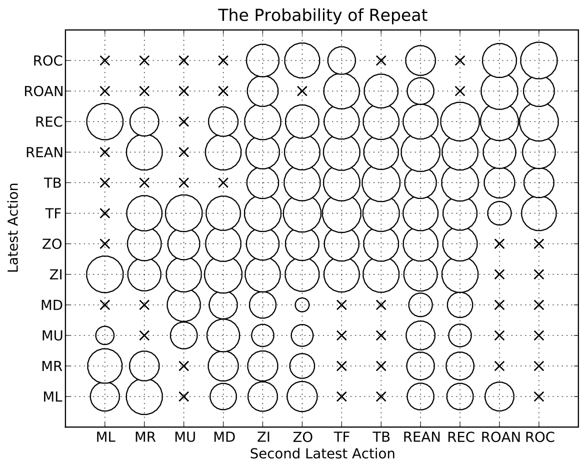
Next, we check whether it helps to consider one more action, the second previous action. The P_{repeat} under different combinations of previous two actions for Thai Mesh is shown in Figure 5.8. In this figure, we can see that the second previous action does affect the P_{repeat}, but the effect is much less significant than the previous action. Moreover, we found that in around 99.9% of the combinations, repeating the previous action still has the largest probability, so the prediction based on previous action and the one based on previous two actions are the same in most of the cases. Therefore, we conclude that considering only the previous action is enough for the prediction.

Considering Viewpoint The other method to improve the prediction accuracy is to consider the current viewpoint of the user. We compute the conditional distribution of next action A_n on each viewpoint, and always choose the action with the highest conditional probability as the prediction.

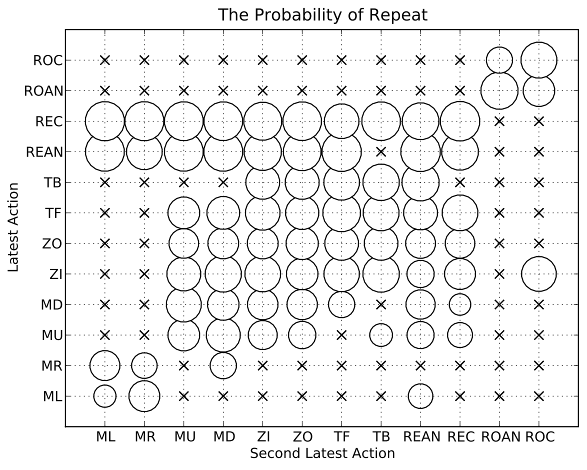
Since we estimate the probability of an action as the frequency of its occurrence in the real traces, enough samples are required to give reliable results. By analyzing the traces we collected, we found that majority of the samples are on a small proportion of viewpoints, and for the rest viewpoints, we have few samples (See Figure 5.9). Including the viewpoints having few samples may exaggerate the accuracy. For example, the predictions on the viewpoints having



Buddha

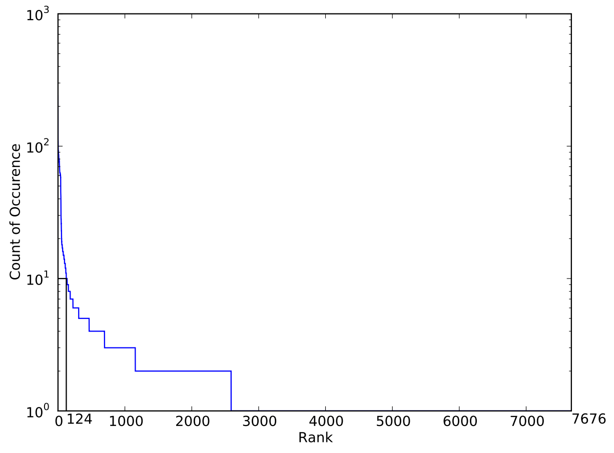


Dragon

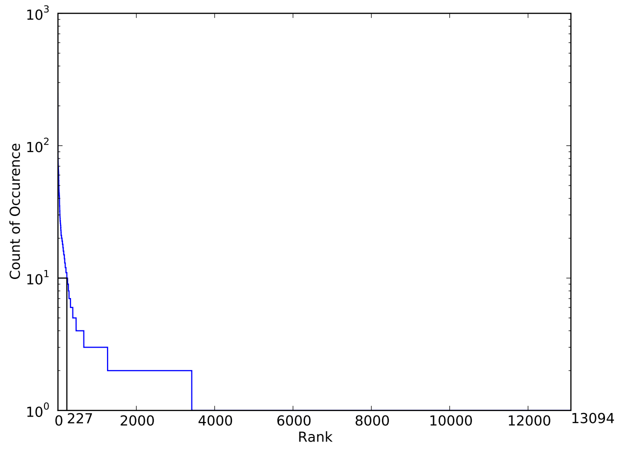


Thai Statue

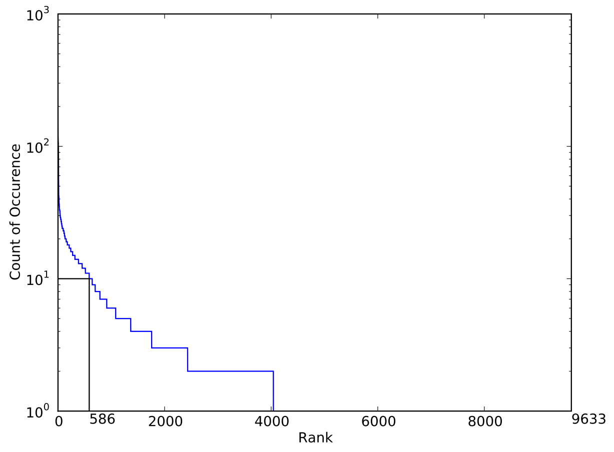
Figure 5.8: The probability of repeating the previous action for each combination of previous two actions (Thai Mesh). 'x' means that the sample size of this combination is too small (less than 10 samples) to obtain reliable probability.



Buddha



Dragon



Thai Statue

Figure 5.9: The number of samples on all viewpoints, and the viewpoints are sorted from the most popular to the least popular.

one sample are always true, which is not realistic. Therefore, we ignore all those viewpoints with less than 10 samples. Figure 5.11 (labelled "Vp") shows the accuracy of this method. We can see that this method has lower accuracy than the one based on previous action.

Considering Viewpoint and Previous Action To improve the accuracy, we could consider both viewpoint and previous action. Similarly, we compute the conditional distribution of next action A_n for each combination. Then we always choose the action with highest conditional probability as the prediction.

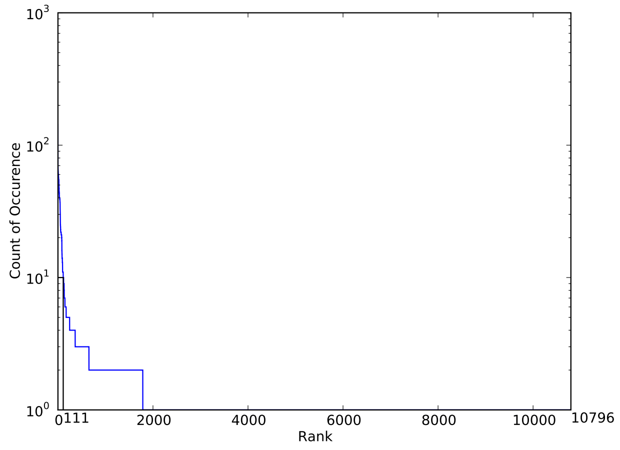
The same problem of lacking samples exists for most combinations (See Figure 5.10). We ignore those combinations with less than 10 samples. The accuracy of this method can be seen in Figure 5.11. This method has the highest accuracy among the four methods.

We compare the four methods in Figure 5.11 for three meshes[2]. We have found that in our scenario the user actions are somewhat predictable and the prediction is simple. Predicting the next action be the same as the previous one without considering any extra information can achieve good enough accuracy in our cases. Considering both previous action and viewpoint has the highest accuracy, but it depends on the statistics of user behavior and differs on every mesh. Hence, it can only be done when enough user traces are already collected for a specific mesh. The slightly better accuracy may not justify the much higher complexity. The interesting finding provides a preliminary evidence that pre-fetching can be useful in mesh streaming.

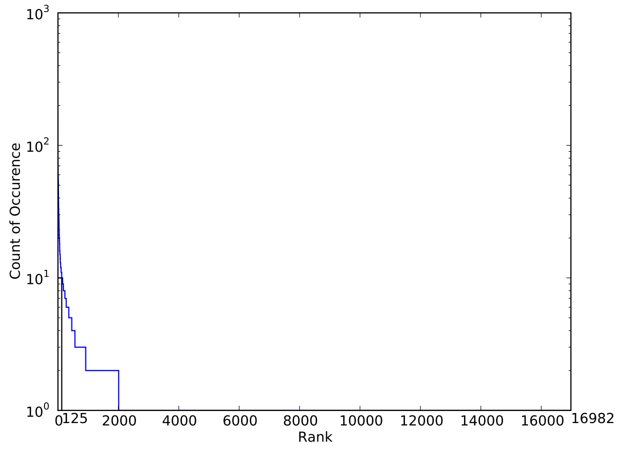
5.3.2 Guiding Caching with User Traces

Caching techniques are used to spend limited resources (such as servers, memories, and network bandwidth) on the most popular requested data to improve the system performance. Hence, how effective a caching system can be depends on the access pattern of users. For example, if most user requests concentrate on a small part of the data, then a small cache can significantly improve the system performance. On the contrary, if all data are equally important, then caching is much less effective. Therefore, to study user traces is crucial in designing a caching system. In this section, based on the analysis of the traces we collected, we show that in our case how caching is useful in three different scenarios: (i) progressive mesh streaming, (ii) remote rendering, and (iii) mesh rendering.

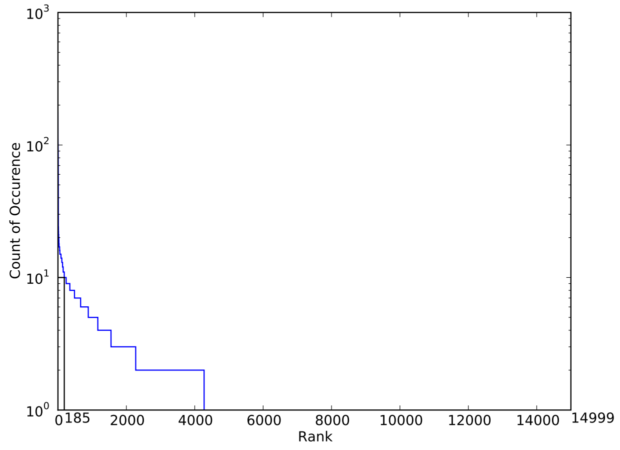
[2]To increase the sample size, we aggregate the data of the three version of the same mesh together. It is reasonable because the meshes having a defect are almost identical to the original one, and users do not know which one has defect in advance. Therefore, their behavior on all three versions of the same mesh are similar.



Buddha



Dragon



Thai Statue

Figure 5.10: The number of samples on all combinations of the viewpoint and previous action, and the combinations are sorted from the most popular to the least popular.

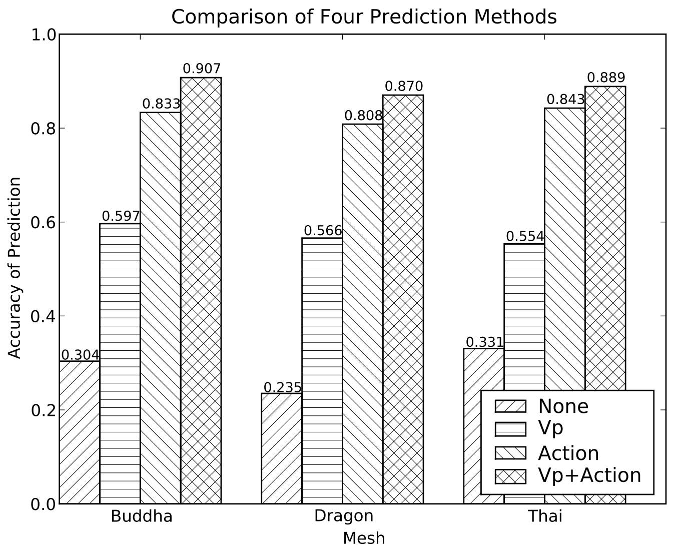


Figure 5.11: The accuracy of four prediction methods.

Caching for Mesh Streaming. In progressive mesh streaming, vertex splits can be cached at a proxy to reduce server overhead. Usually, caching proxy has limited memory (or it can only provide limited memory for a certain mesh), so only a small part of the vertex splits can be stored. To study the usefulness of proxy caching, we look at the access pattern to vertex splits.

We first replay the log of actions from the users, and generate a list of vertex splits accessed by users during the experiment. From these vertex split requesting traces, we count the number of times each vertex split is accessed. We then sort the vertex splits in decreasing order of the access count, and plot the cumulative access count versus rank in Figure 5.12(a). We normalize both axes to between 0 and 1 so that we can plot all three meshes on the same graph. Figure 5.12(a) shows how many requests (in percentage) we can satisfy (i.e., hit rate) by storing the most frequently requested chunks in a proxy. The x-axis denotes the number of vertex splits stored in the proxy, as a fraction of total number of vertex splits requested (note: not the total number of all vertex splits). It can be observed that by building a static cache that stores 20% of the most frequently accessed vertex splits, the proxy can achieve more than 70% hit rate for *Thai Statue* and *Dragon*, and 55% for *Happy Buddha*.

Caching of Remote Rendering. For rendering on mobile devices [12] or for protection of mesh data [51], the server could send a rendered image directly according to the users' viewpoint.

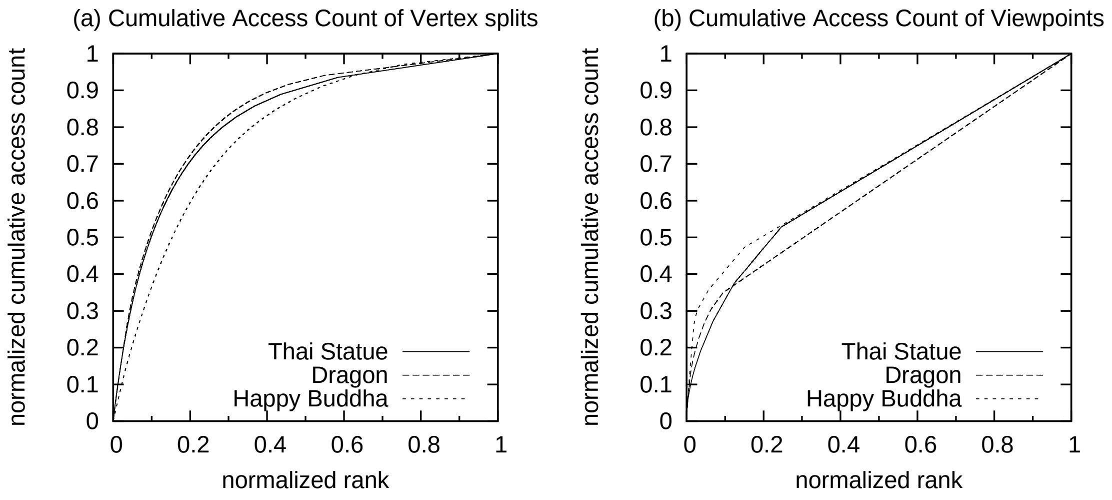


Figure 5.12: Cumulative Access Count versus Rank

In this scenario, the caching proxy can cache the rendered images. We can similarly find the viewpoints “visited” by the most users. A viewpoint visited multiple times by the same user is only counted once, since the user can keep the received image locally and need not request it the second time. Figure 5.12(b) shows a plot similar to Figure 5.12(a), but for access frequency of viewpoints. The figure shows the hit rate at the caching proxy if we choose to store pre-rendered images corresponding to the most frequently accessed viewpoints. The distribution is not as skewed as the access count for chunks, but still, caching the rendered images for 20% of the most frequently accessed viewpoints can yield 40 - 50% hit rate in our scenario.

Caching of Vertices and Pixels. Caching mesh data in graphic card memory (e.g. using VBO (Vertex Buffer Object) and PBO (Pixel Buffer Object) supported in OpenGL), could significantly increase the rendering speed when the memory bandwidth is the bottleneck. For graphic cards without enough memory to store the whole mesh, we could just store the most frequently viewed part of the mesh in the graphic card memory.

We replay all the user traces, and whenever the viewpoint changes we increase the access count of the visible faces by one. Therefore, the access count of a face is the count of viewpoints at which it is visible (revisiting to a previously visited viewpoint is also counted since the mesh will still be rendered). We normalized the number of views of each face and visualize them with a heat map (Figure 5.13). We can see that the most frequently viewed region of *Happy Buddha* (viewed 4205 times) is the base between the two legs because it is visible from both the front and the back. Figure 5.14 plots the normalized cumulative view count of faces versus rank, similar to Figure 5.12. We can see that the locality is slightly less than that in the previous two scenarios, but for *Happy Buddha* and *Thai Statue*, a hit rate of 40% can be achieved by storing 20% of the most frequently viewed faces in the graphic card memory. The mesh *Dragon* has the least locality. We hypothesize that this is because people tend to view *Dragon* at many different viewpoints due to its complex shape, leading to more evenly distributed viewpoints around the mesh.

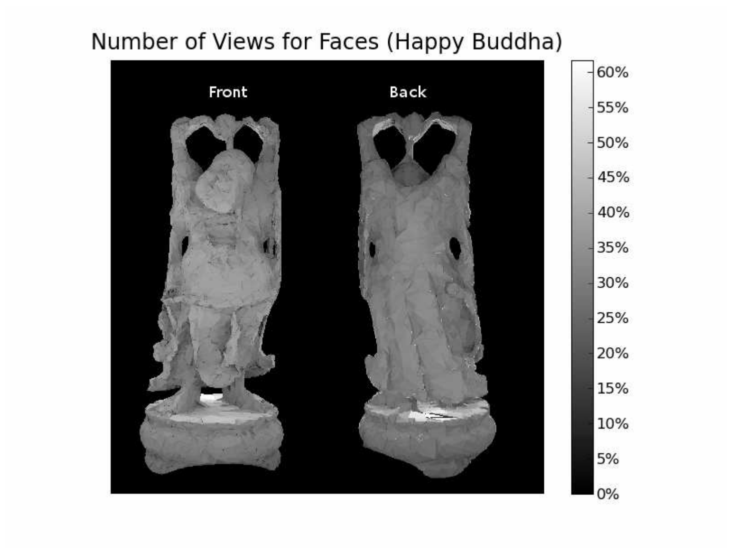
5.4 Generating Synthetic Traces

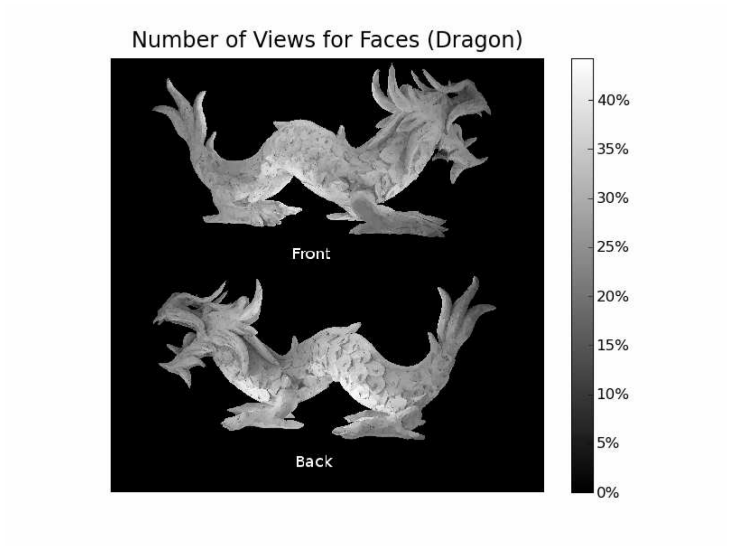
To evaluate the effectiveness and performance of a system, we often need many user traces. It is expensive to collect a large number of real traces. A cheaper way is to consider a user trace as a Markov chain and derive the transition matrix from the collected traces. Then following the transition matrix, we can generate as many traces as we want.

As we introduced in Section 5.2, a user session is a random process with viewpoints as the states. Each action transits the system from one state to next state, until the session is terminated (see Figure 5.4 as an example). To generate a synthetic trace, we first need to determine the session length, i.e. how long the user spent in viewing on this mesh. Then we begin from the default viewpoint and keep determining the next action until the session time is expired. For each action, we need to determine two things: when and what the next action is. In this section we first introduce how we determine the session length and the interval between two actions, named *think time*, and then we introduce how we choose the next action based on current state in detail.

5.4.1 Session Length

Session length refers to the time each user spends in viewing a mesh. Generally, the session length is short, with the average values of 107s, 76s, 47s for *Thai Statue*, *Dragon*, and *Happy Buddha*, respectively (see Table 5.3). Figure 5.15(a) shows the distribution. The session length decreases with complexity of the meshes, as expected. The session length fits the *log-normal* distribution. For example, the session length of Thai Statue follows a log-normal distribution with $\mu = 18.23$, $\sigma = 0.754$ and the unit is μs. When we generate a synthetic trace, we determine





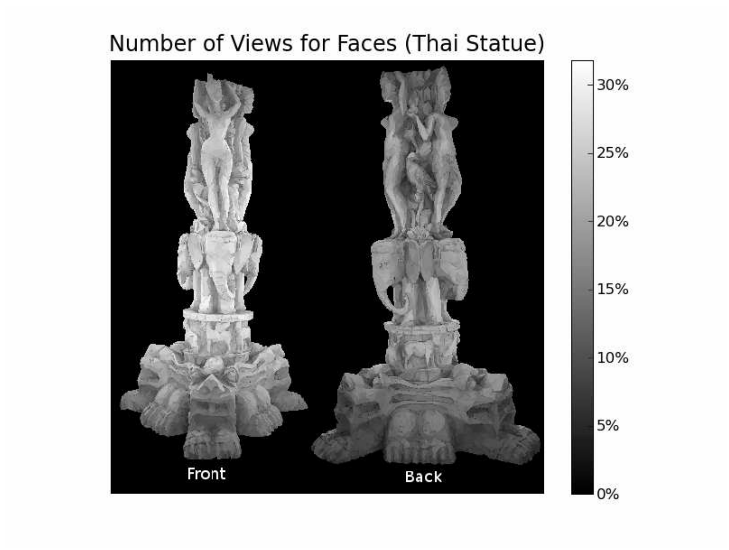


Figure 5.13: The normalized number of views for each face.

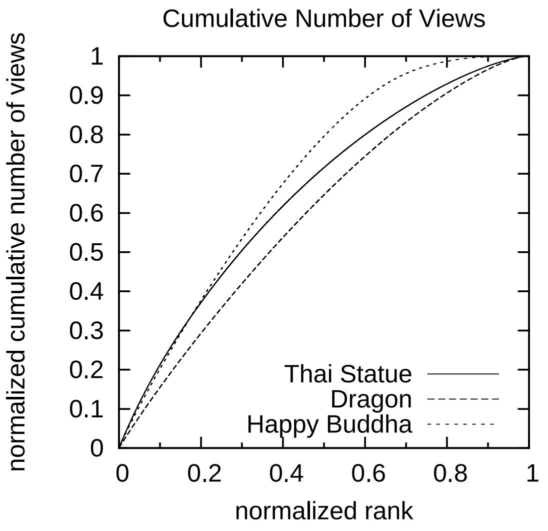


Figure 5.14: The hit rate when we save part of the faces in the graphic card.

the session length by generating a random number following the log-normal distribution with parameter derived from the real traces.

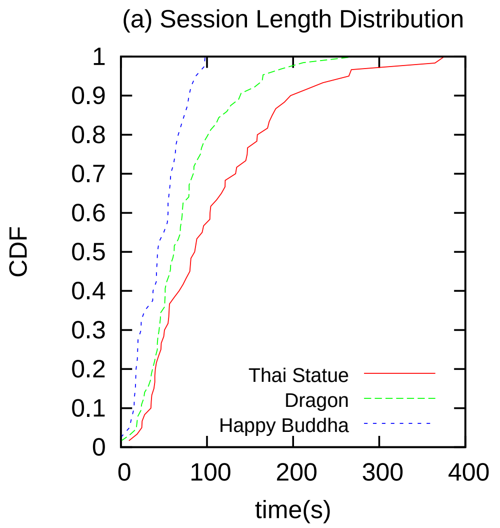


Figure 5.15: Distribution of Session Length

Mesh	Session Length		Think Time	
	Mean(s)	Max(s)	Mean(ms)	Max(s)
Thai Statue	107	376	593	25
Dragon	76	272	574	20
Happy Buddha	47	98	403	13

Table 5.3: Session Length and Think Time

5.4.2 Think Time

We refer to the time between two actions as *think time*. We find that think time follows similar distributions for all of the three meshes (Figure 5.16(a)). The mean and maximum think time are shown in Table 5.3. We note that the maximum is up to 50 times larger than the mean, but about 90% of the think time is smaller than a second.

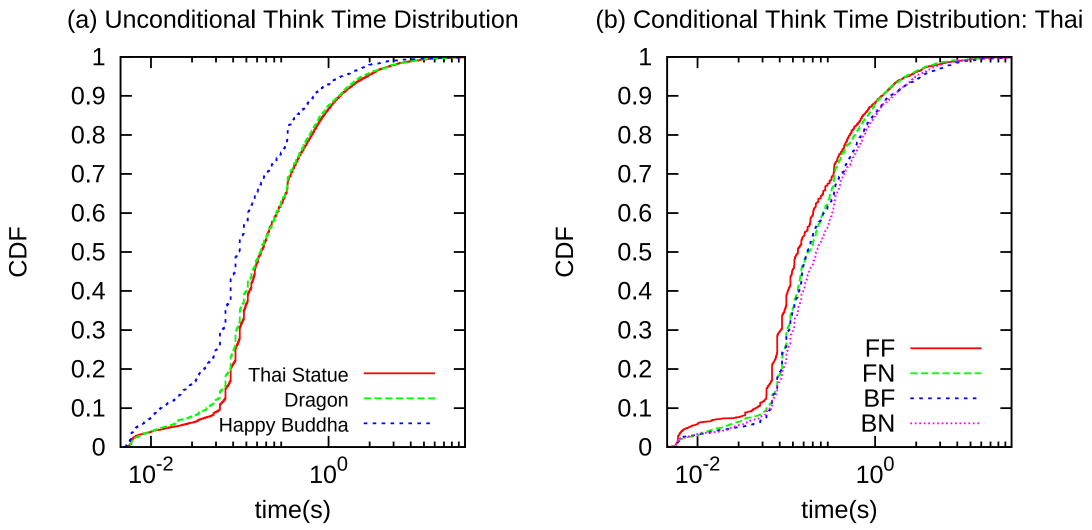


Figure 5.16: Think Time (x-axis in log scale).

A simple method to determine the think time is following the distribution we derived from the traces, without considering the current state. This may be not accurate since the current viewpoint may affect the distribution of the think time. Furthermore, people tend to repeat the same action quickly, and usually take longer time if they change to another action. Therefore, whether the next action is repeating the last action is also a factor to be considered.

To investigate the relation between think time and viewpoints, we classify the viewpoints into 4 regions: front-far (FF), front-near (FN), back-far (BF), and back-near (BN), based on *front/back* and *far/near*. We find that the effect of region on think time distribution is not apparent 5.16(b)). Hence, for simplicity we ignore the effect of viewpoints during determining think time.

Next, we examine the effect of repeating or non-repeating the previous action. Figure 5.17 shows think time tends to be much shorter when users repeat the previous action. Hence,

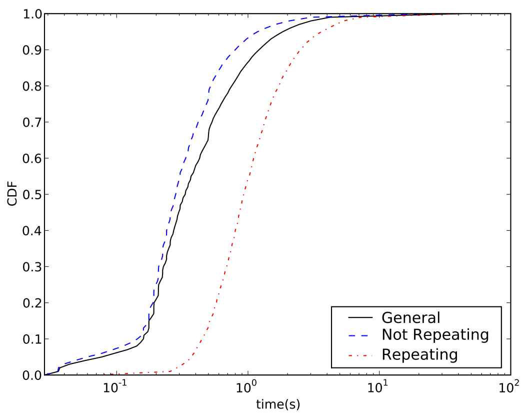


Figure 5.17: Think time is smaller when user repeat the previous action.

during the synthetic trace generating, we use the respective distribution after we decide the next action. Next, we introduce how we determine the next action.

5.4.3 Next Action

We decide the next action based the Markov chain method. The viewpoint, represented as a 6-tuple: $(x, y, z, \theta_x, \theta_y, \theta_z)$ as introduced in Section 5.2, can be defined as the states in the Markov chain. According to the observation that the previous actions highly affect the next action (Section 5.3.1), it is not a first order Markov chain because the previous states affect the transition probability. It, however, is not practical to consider the whole history before. To make the analysis possible, we consider only one step before and assume "(previous state, current state)" decides "next state". In other words, it is a second order Markov chain. As we introduced in Section 5.3.1, the latest action affects the most, so this simplification is reasonable.

We can reduce the second order Markov chain to first order by defining the current state as (previous viewpoint, current viewpoint). After a user action, the new state becomes (current viewpoint, next viewpoint) (See Figure 5.18(a)). Then, the transition probability only depends on the current state. Notice that (previous viewpoint, current viewpoint) is equivalent to (current viewpoint, previous action), so we finally define the state to be $(x, y, z, \theta_x, \theta_y, \theta_z, A_p)$. (See Figure 5.18(b)).

Each record in the user traces can be seen as a sample of this random process. From these samples, we can compute the transition probability between two states, and hence the transition matrix, with which we can generate synthetic traces.

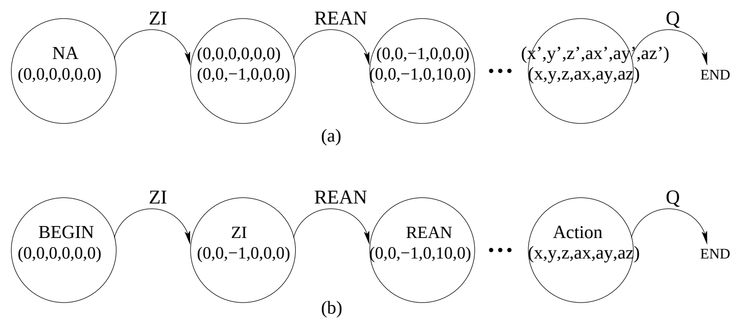


Figure 5.18: (a), State = (previous viewpoint, current viewpoint); (b), State = $(x, y, z, \theta_x, \theta_y, \theta_z, A_p)$

This method, however, has limitations when the number of collected traces is small. First, if a viewpoint is never accessed by the real traces, it will not accessed by synthetic traces either. Therefore, no matter how many synthetic traces are generated, the number of accessed viewpoints is upper-bounded. Second, due to the large sample space and limited number of samples, most of the states have only one sample (See Figure 5.10). In other words, in the transition matrix, the next action for most states is determined. As a result, after several steps, the synthetic trace follows the exact path as one of the real traces. Therefore, the synthetic traces are not random enough.

To address these two limitations, we propose a method to generate traces that are more random in the next section. It is based on a simplified model, which does not directly derive and follow the transition matrix.

5.4.4 Simplified Model

To generate a synthetic trace, we need to determine the probability of each possible A_n based on current state $(x, y, z, \theta_x, \theta_y, \theta_z, A_p)$. Due to the limited number of samples, we cannot obtain the reliable transition probability for most of the states. As a result, we need to find a new way to determine the conditional distribution of A_n.

To address the problem of lacking samples, we consider only the main factors affecting the probability of A_n and ignore others. According to our observation, repeating the previous action has the highest probability in most cases. Therefore, the previous action is chosen as one of the most important factors. Based on the relation to the previous action, we categorized actions to three groups: *repeat*, *reverse*, and *change*. "Repeat" means the next action is the same as the previous action; "reverse" means the next action is in the opposite direction of the previous action. For example, if the previous action is "Move Up", then "reverse" means the next action is "Move Down". All other actions are categorized as "change" action.

We determine the next action in two steps:

1. We determine the probability of "repeat", "reverse", and "change", i.e.

$$\begin{aligned}
P_{repeat} &= Pr(A_n = A_p|_{\text{current state}}) \\
P_{reverse} &= Pr(A_n = \text{reverse}(A_p)|_{\text{current state}}) \\
P_{change} &= Pr(A_n! = A_p, A_n! = \text{reverse}(A_p)|_{\text{current state}}),
\end{aligned}$$

2. We decide the distribution of A_n if "change" is decided, i.e. $Pr(A_n|_{\text{current state},A_n!=A_p,A_n!=\text{reverse}(A_p)})$ for each possible A_n.

In Step 1, in addition to the most important factor, the previous action, we consider only one coordinate instead of six. As we know, an action only changes one coordinate in the 6-tuple. We call this coordinate *corresponding coordinate*. For example, the corresponding coordinate of "Move Left" is x, and the corresponding coordinate of "Revolving Clockwise" is θ_y. When users are repeating or reversing the previous action, only the corresponding coordinate is changed. As a result, we only consider the corresponding coordinate in determining the P_{repeat} and $P_{reverse}$. For example, when a user keeps pressing "Move Left", the mesh may nearly be out of the viewable area, then the P_{repeat} decreases, and the $P_{reverse}$ increases. Another example is that when the user keeps pressing "Revolve Clockwise" and find something interesting in the mesh from the current direction, the P_{repeat} decreases and the P_{change} increases (the user may "Zoom In" to see more details or "Move" to the interesting part). In summary, in this simplified model, we assume that P_{repeat}, $P_{reverse}$, and P_{change} only depend on (previous action, corresponding coordinate). As a result, the number of available samples of each combination increases a lot compared with the first method (See Figure 5.19).

In Step 2, we need to determine $Pr(A_n|_{\text{current state},A_n!=A_p,An!=\text{reverse}(A_p)})$ for each possible A_n. In this step, we ignore the effect of previous action. We think that $P_{continue}$ is high mainly because users tend to press the same key multiple times. It affects the P_{change}, but its effect on which action to be chosen as "change" is little. Hence, we assume that

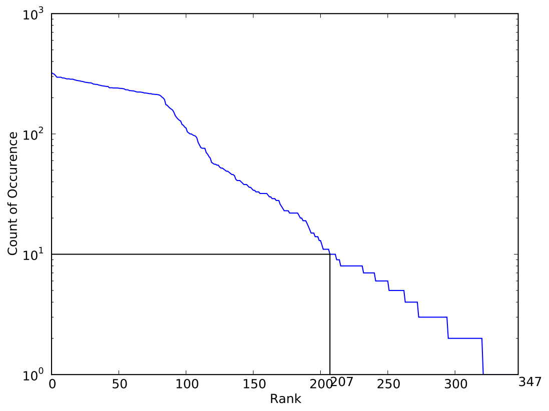


Figure 5.19: The sample size of each combination (previous action, corresponding axis), sorted from the most popular to least popular. Mesh: Thai

$Pr(A_n|_{\text{current state},A_n!=A_p,An!=\text{reverse}(A_p)})$ only depends on $(x, y, z, \theta_x, \theta_y, \theta_z)$. Here, all coordinates, except the corresponding coordinate (it will not be changed since "repeat" and "reverse" are already excluded from the choices), are important to decide the next action. We cannot directly derive the probability from the traces we collected because the sample size is not large enough. Instead, we propose a heuristic based on the assumption

$$Pr(x, y, z, \theta_x, \theta_y, \theta_z) = Pr(x)Pr(y)Pr(z)Pr(\theta_x)Pr(\theta_y)Pr(\theta_z)$$

In other words, we assume that the probability for a viewpoint to have a specific coordinate is independent of its other coordinates. This is an assumption to simplify our model, and will introduce some difference between synthetic traces and real traces. In the validation part, however, we show that the difference is not significant.

At a specific viewpoint Vp: $(x, y, z, \theta_x, \theta_y, \theta_z)$, if we change the value of x, then we stay in the 5-dimensional space

$$(Y = y, Z = z, AX = \theta_x, AY = \theta_y, AZ = \theta_z).$$

The probability of the viewpoint staying in this space is

$$\begin{aligned}
&Pr(y, z, \theta_x, \theta_y, \theta_z) \\
=&Pr(y)Pr(z)Pr(\theta_x)Pr(\theta_y)Pr(\theta_z) \\
=&\frac{Pr(x)Pr(y)Pr(z)Pr(\theta_x)Pr(\theta_y)Pr(\theta_z)}{Pr(x)} \\
=&\frac{Pr(x, y, z, \theta_x, \theta_y, \theta_z)}{Pr(x)} \\
=&\frac{Pr(Vp)}{Pr(x)}.
\end{aligned}$$

Similarly, we can know the probability of the viewpoint staying in other spaces.

Hence, we set the probability of changing a value of an axis to be proportional to the probability of staying in the corresponding space. For example, the probability of changing x when the previous action is on AY

$$\begin{aligned}
=&\frac{\frac{Pr(vp)}{Pr(x)}}{\frac{Pr(vp)}{Pr(x)} + \frac{Pr(vp)}{Pr(y)} + \frac{Pr(vp)}{Pr(z)} + \frac{Pr(vp)}{Pr(\theta_x)} + \frac{Pr(vp)}{Pr(\theta_z)}} \\
=&\frac{\frac{1}{Pr(x)}}{\frac{1}{Pr(x)} + \frac{1}{Pr(y)} + \frac{1}{Pr(z)} + \frac{1}{Pr(\theta_x)} + \frac{1}{Pr(\theta_z)}}.
\end{aligned}$$

Similarly, we can know the probability of changing other coordinates. Intuitively, we can think of $Pr(x)$ as the popularity of the current x value. The above equation means that we tend to change the coordinate having low popularity.

Next, for each axis we have two directions to move, and we need choose one. We assume users tend to move to a direction with higher popularity, i.e. $Pr(vp)$. We define the popularity of a direction to be the sum of popularity of several consecutive positions in that direction (we choose 3 positions in our implementation). Then, we assume the probability to go in one direction is proportional to the popularity of that direction. For example, if the axis to change is X and its current value is x, the probability of moving right (considering three consecutive positions in that direction: $x + 1$, $x + 2$, $x + 3$)

$$\begin{aligned}
=&\frac{\text{Popularity(right)}}{\text{Popularity(right)} + \text{Popularity(left)}} \\
=&\frac{Pr(x+1) + Pr(x+2) + Pr(x+3)}{Pr(x+1) + Pr(x+2) + Pr(x+3) + Pr(x-1) + Pr(x-2) + Pr(x-3)}.
\end{aligned}$$

Notice all these viewpoints have the same y, z, θ_x, θ_y, and θ_z values, so they are not in the equation. We consider several positions instead of one because users tend to repeat the same action several times, so once a direction is chosen, the following several positions instead of one could be visited.

In brief, we generate a synthetic trace as follows:

1. Determine the session time according to the distribution of session time we derived from the real traces.

2. For the default viewpoint, we derive the transition probability from the traces and decide the next action based on it.

3. For other viewpoints, we choose "repeat", "reverse", or "change" based on previous action, corresponding coordinate.

4. If we choose "change", we first determine the axis to change based on $Pr(x)$, $Pr(y)$, $Pr(z)$, $Pr(\theta_x)$, $Pr(\theta_y)$, and $Pr(\theta_z)$. Next, we choose the direction based on the popularity of three neighbors in both directions.

5. We randomly select a think time from two different distributions based whether the action is "repeat" or not (see Section 5.4.2). Then go back to step 3 unless the session time has expired.

Based on our observations, the synthetic traces generated by our method satisfy some important requirements. When replaying the traces we generated, the mesh will not stay in abnormal state for long (abnormal positions include upside down, tilted, out of viewable area, etc.). Since we determine P_{repeat} and $P_{reverse}$ considering the corresponding coordinate, the probability for the mesh to enter abnormal state is low, and it tends to go back to normal state by "reverse" action. Moreover, whenever "change" is chosen, the mesh tends to go to a more normal state (a state having higher probability).

5.4.5 Comparison

In this section, we compare the synthetic traces we generated with the real traces. We generated 10000 synthetic traces following the simplified model we introduced in the last section. Thai mesh is chosen as the mesh used in this validation because of its high complexity. Thai statue is the largest mesh and has many details, so more variety exist in users interactions.

First, we compare the unconditional probability (the probability without consideration of any condition, such as the current viewpoint or the previous action) of each action in the traces. From Figure 5.20, we can see that the probability of each action in the traces we generated fits well with that in the real traces. This result is impressive because we never consider unconditional probability in our model, but still the traces generated have similar unconditional probability for each action.

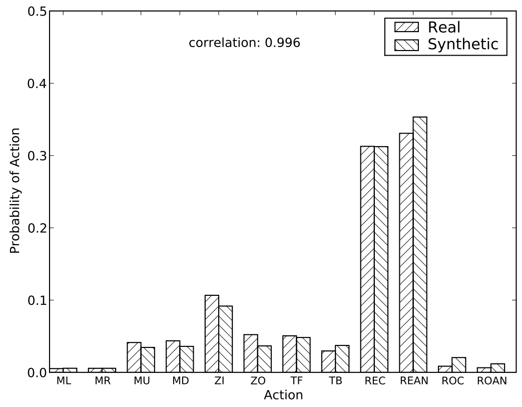


Figure 5.20: Comparison of occurrence frequency of each action.

Second, we compare the P_{repeat}, including unconditional P_{repeat} and conditional P_{repeat} under different previous action. We can see from Figure 5.21, the synthetic traces we generated have similar P_{repeat} under all conditions.

Third, we compare how the viewpoints distributed on each axis, i.e. the value of $Pr(x)$, $Pr(y)$, $Pr(z)$, $Pr(\theta_x)$, $Pr(\theta_y)$, and $Pr(\theta_z)$. Figure 5.22 shows that the synthetic traces we generated has similar distributions on all axes. Among the six axes, only the Y axis has a slightly higher error. The synthetic traces tend to stay in the center (0) more often. The correlation between these two sets of values, however, is still close to 1. Again, this result is interesting. Although we have considered these probabilities in selecting new action when "change" is selected, the majority of selections (recall that more than 80% of actions are "repeat") has nothing to do with these values.

The final comparison is on the transition probability on certain state, $Pr(A_n|_{(x,y,z,\theta_x,\theta_y,\theta_z,A_p)})$. If the Markov chain method is used, the synthetic traces should have the same transition prob-

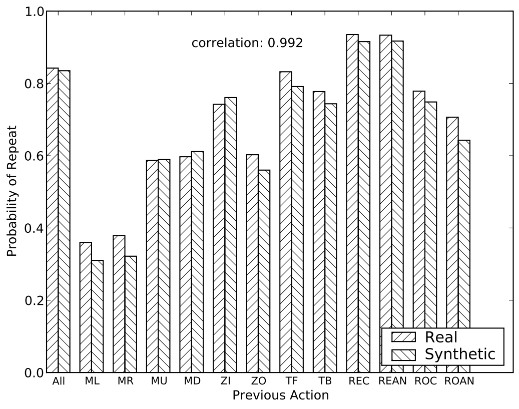


Figure 5.21: Comparison of unconditional repeat probability and repeat probability under different previous actions.

ability as the real trace. Due to the limited sample size, however, we have to use a simplified method. This comparison shows the difference introduced by this simplification. We only choose some states with a sufficient number of samples to ensure the transition probabilities are reliable. Some of the results are shown in Figure 5.23. In most of the cases, the synthetic traces have similar transition probability. The largest error happens in the state (0, 0, 12, 0, 0, 0, ZOOM IN) when the sample size is small (22) and the distribution of next action is relatively even. In this case, the transition probability derived from the real traces itself is not so reliable.

In summary, due to the limited number of real traces, we cannot completely validate the synthetic traces we generated by checking the transition probability on each state. According to the comparison we did, however, the synthetic traces we generated are similar to the real traces. Moreover, according to our observation, the traces we generated have much less abnormal states (upside down, tilted, out of viewable area, etc.) than the random trace. Therefore, we think it may be more realistic to use these synthetic traces rather than random traces to test the performance of our mesh streaming system.

5.5 Conclusion

In this chapter we introduced how we generate synthetic traces based on real traces we collected. These synthetic traces are similar to the real traces in the properties we compared. Hence, we use them instead of random traces to measure the performance of the peer-to-peer mesh streaming system introduced in the next chapter.

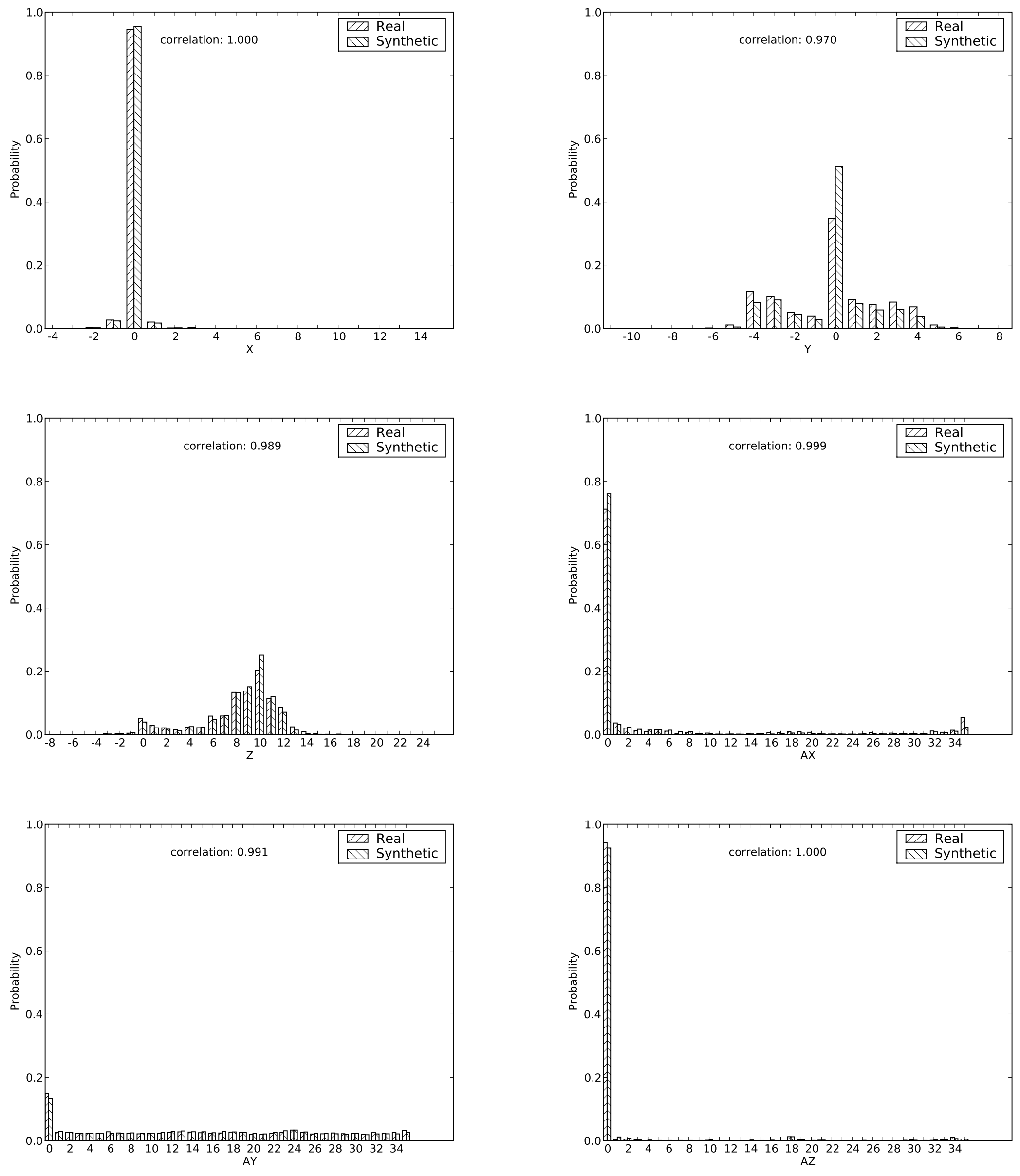


Figure 5.22: Comparison of distribution of viewpoints on different axes.

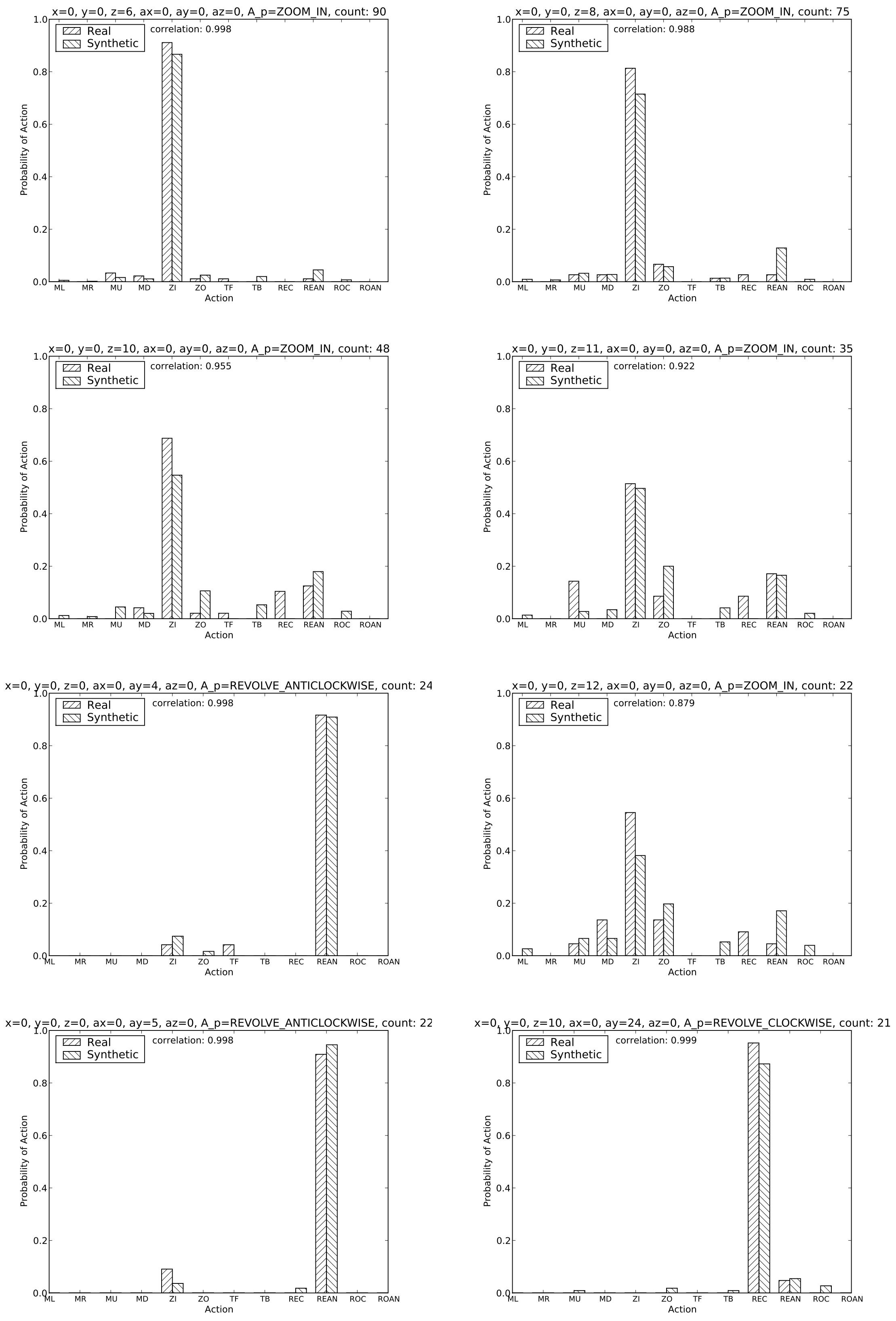


Figure 5.23: Comparison of transition probabilities on certain states.

With a preliminary analysis of the real traces we collected, we have obtained some interesting findings. First, in our usage scenario, user actions are predictable to certain extent. The prediction based on previous action is simple and effective. Therefore, it indicates pre-fetching is possible both in streaming and rendering within our scenario. Second, our analysis reveals that in the traces we collected, locality exists in both data and viewpoint access. It may be possible to exploit this locality in designing progressive mesh streaming systems and rendering systems.

While our original goal is to gather user traces to evaluate our system, our findings above, despite being limited to a single usage scenario, points to a potentially important new research topic. Our study in this thesis establishes the first step towards that direction.

Chapter 6

Peer-Assisted View-Dependent Streaming

6.1 Introduction

In applications such as virtual art gallery, virtual earth, virtual museums, and virtual auction house, statues, artifacts, and auction items are streamed to potentially a large number of visitors or bidders, sometimes within a short period of time (e.g., when an item is released for bidding). The clients for these applications are likely to inspect the items carefully, zooming in to view the fine details, and rotating the item back-and-forth to examine all facets of an item. Such flash crowd can potentially impose substantial bandwidth requirements on the server.

Peer-to-peer (P2P) data dissemination is commonly used to alleviate server's bandwidth cost. Downloading peers forward the received data to other peers, contributing their upload bandwidth and reducing the burden of the server, hence allowing more users to be supported. P2P techniques have been successfully used in bulk file transfer (e.g., BitTorrent) and video streaming (e.g., PPLive).

Using P2P techniques for view-dependent progressive mesh streaming, or *P2P mesh streaming* for short, poses several new challenges. First, each client may view, and therefore request mesh data, in a different order. Thus, a client needs to frequently search for peers to download visible mesh data from when its view point changes. Second, we found that a user stays in the system for a short time, in the order of minutes (See Section 5.4.1) when viewing a mesh, leading to high churn rate. Third, determining the visible region of a mesh given a view point is computationally intensive and is traditionally done at the server. In the P2P context, how to determine the visible region of a mesh is non-trivial.

Fortunately, the nature of mesh streaming alleviates other constraints in the system design: (i) we found that users can tolerate higher response time when interacting with the mesh, due to progressive mesh rendering, and (ii) unlike video, there is no strict deadline to render the mesh. Together, these differences lead us to new challenges and different design decisions in designing P2P mesh streaming systems, in contrast to P2P video streaming and P2P file downloading. This chapter reports on our design of a P2P mesh streaming system and its evaluations.

In P2P data dissemination, the data (a file, video, or mesh) are typically divided into *chunks*. A peer that has downloaded a chunk can potentially serve this chunk to other peers. We call such a peer a *provider* of the chunk. This chapter addresses two important design questions of P2P mesh streaming.

The first question is how to define a chunk. While one can easily segment a file or a video into chunks, due to progressivity of progressive meshes, chunks have to be carefully constructed in a hierarchical way that can incrementally improve the quality of the mesh, with minimal coding dependency among the chunks. Moreover, chunk size should be small enough to reduce the fraction of invisible vertex splits received.

Second, how can a peer know the best provider of a given chunk? Consider the a large number of queries and high churn rate, we first consider simply using a centralized lookup service for chunk provider, allowing fast peer failure detection and one-hop lookup. Centralized lookup, however, is not scalable to large number of peers, due to the overhead in maintaining peer states and handling queries. We therefore propose a hierarchical P2P system, which retains the advantages of centralized lookup but with significantly fewer requests to the server. The basic idea is to group peers according to the hierarchical structure of chunks in a progressive mesh. Each group has a leader that takes over most of the responsibilities of the server to reduce server overhead.

Our contributions in this chapter are as follows. First, we compare and contrast P2P mesh streaming to P2P video streaming and P2P file downloading and point out the main difficulties of P2P mesh streaming. Second, considering the differences and challenges, we investigate two content discovery schemes for P2P mesh streaming. The structure of the progressive mesh is considered in the second design to further reduce server overhead. Third, we propose a chunking scheme for progressive meshes to support P2P mesh streaming. Finally, we run simulations based on synthetic traces generated with the method introduced in Chapter 5 to evaluate the P2P mesh streaming system we proposed. Analysis and simulation results show that server overhead can be reduced by 90%. Meanwhile, average response time and control overhead are low.

We structure the rest of this chapter as follows. We compare P2P mesh streaming with P2P video streaming and P2P file downloading in Section 6.3. Section 6.4 introduces the hierarchical chunk structure. We propose two content discovery schemes in Section 6.5, and the experimental results in Section 6.6. Finally, we conclude in Section 6.7.

6.2 P2P Video Streaming

P2P techniques have been widely studied in file downloading, live streaming, and video-on-demand (VoD) streaming applications. In this section, we introduce the related work on P2P video streaming systems.

Generally, P2P streaming systems can be categorized into tree-based and mesh-based approaches. In tree-based approaches, peers are organized into a single tree [43, 25] or multiple trees [15, 62]. In typical multi-tree-based approaches, like CoopNet [62], different parts of a video segment are separately pushed down along different sub-trees. SplitStream [15] utilizes Multiple Description Coding (MDC) to further improve streaming quality by introducing data redundancy. Besides, each peer serves as an interior node in only one sub-tree to minimize the negative effect of node failures.

To further improve robustness and scalability, mesh-based approaches have been more widely explored in P2P live streaming systems, including PPLive and PPStream. Typically, gossip-like protocols are used for peers to pull video content from qualified peers. Compared to tree-based approaches, such systems incur higher end-to-end latency and control overhead. To reduce these negative effects, push-pull-based approaches are proposed in [90]. Moreover, PRIME [57] incorporates a swarming scheme and MDC to improve robustness and streaming quality, but its two-phase design increases end-to-end latency.

Similar tree-based and mesh-based approaches can also be utilized in P2P VoD streaming. Different from live streaming, peers may seek to different playback points in the video, causing higher peer dynamics. Further, since peers have different views of the video, it is more challenging to locate providers. The oStream [28] system utilizes sliding window to maintain certain video content so that peers can obtain content from other peers with smaller interval between their windows. Yiu et al. [88] propose that each peer randomly stores some segments and uses DHT to locate peers storing previous, current, and next segments, respectively. This approach can achieve better scalability and shorter initial delay than a sliding window scheme, but maintaining the segment lists in a highly dynamic environment is difficult.

6.3 P2P Mesh Streaming

Although P2P techniques have already been studied in file downloading, live video streaming, and video-on-demand streaming, P2P view-dependent progressive mesh streaming (or P2P mesh streaming for short) has a different set of requirements and characteristics, leading to different design decisions and new challenges. In this section, we elaborate on these differences.

In P2P file sharing, a peer is interested in obtaining a complete file. In most cases, the file is not useful until it is completely downloaded. During downloading, *any* new chunk of the file downloaded is useful. Chunks can be downloaded in any order. Therefore, peer p can receive chunks from peer q as long as q possesses fresh chunks that p has not received.

In P2P video streaming, however, the video is played back mostly in increasing time order[1]. The point the peer is watching the video is the *playback time*, and each chunk in the video has an associated timestamp. Unlike file sharing, not every chunk is useful. A peer is only interested in a chunk whose timestamp is later than the playback time. Chunks nearer to the playback time have higher priority. Since chunks are needed and playback in the same order, if a peer p receives a chunk with timestamp t from a peer q, it is likely that p can receive the chunk with timestamp $t+1$ from q. In other words, a peer can exploit *temporal locality* in chunk access to discover other peers to retrieve the chunks from.

Temporal locality does not apply to meshes. In P2P mesh streaming, however, users might be interested in different parts of the mesh at different level of details. Each may choose to

[1]unless the user seek to another playback point

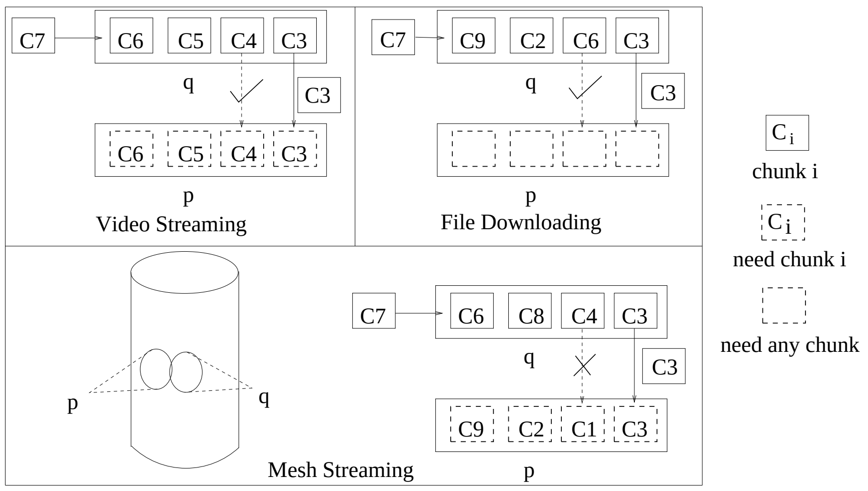


Figure 6.1: In video streaming, peers receive chunks in the same order, so p may request many chunks from q (e.g, in the order of C3, C4, C5). In file sharing, peers can receive chunks in any order, so p can request all chunks received by q (e.g., in the arbitrary order of C3, C6, C2). In P2P mesh streaming, however, a peer often needs to keep finding new providers. If p changes its view to be different from q, q may not be able to supply p with the needed chunks anymore (e.g., C1).

look at different facets, or zoom into different levels. Therefore, each peer may be interested in different chunks of the mesh at different times. Consequently, even if peer p can receive a chunk from peer q now, p may not receive subsequent chunks from q later, since p may request a chunk that q has never seen and is not visible to q. Thus, a peer would have to continuously query for peers to retrieve the mesh data from, as the peer's view point and level of detail change over time. More queries are needed in mesh streaming than in file downloading or video streaming. Reducing such overhead is one of the main challenges of P2P mesh streaming.

Another characteristic of mesh streaming is that the session length, i.e. how long a user stays in the system, is significantly shorter than that in file downloading and video streaming. Users usually leave the system after viewing all the interesting parts of a mesh in several minutes. Such short session time increases the churn rate and makes typical approaches to reduce query overhead inappropriate. To reduce the number of queries, a peer typically caches information about the chunks available in a provider for future requests. High churn rate invalidates this cache information quickly.

P2P view-dependent mesh streaming remains a new topic because of the difference we stated above. Some related studies exist, but they are not sufficient. Hu et al. [42] and Cavagna et al. [16] have considered using peer-to-peer architecture to stream a 3D scene to improve scalability.

Our work is similar in spirit, but we focus on streaming a single, large, progressive mesh in a view-dependent manner, rather than a 3D scene, where visibility decision is mainly done at the object level. The streaming of a single visible object, however, is still following a fixed pre-decided order independent of the user's view point. The object level view-dependent system works well for scenes with many small objects but not for the scenarios we target, where the data size of one object is already huge. Streaming progressive meshes needs a much finer granularity for view dependency. In this chapter, we study view-dependent streaming at chunk level, which can be of much smaller size (can be as small as a packet) than an object.

Yang et al. [87] proposed a view-dependent 3D video streaming system, in which a set of gateways is used to disseminate the streams depending on the views of the receivers. Similar to our goals, the technique aims to reduce the server overhead. The number of gateways, however, is limited and the gateways are assumed to be stable. Further, 3D videos are not progressive in nature.

6.4 Hierarchical Chunk Structure

We now elaborate on how to group vertex splits into chunks while maintaining progressiveness among the chunks. In the original design of the receiver-driven protocol [21], the receiver explicitly requests individual vertex splits. Such design is not appropriate for P2P streaming for three reasons. First, the receiver needs to send one ID (2 bytes in our implementation) to request one vertex split (less than 5 bytes in our implementation). Hence, the requests occupy a large proportion of the up-link bandwidth, which is precious in P2P streaming, in which the up-link bandwidth is needed to share data with other peers. Second, data packets need to be generated dynamically at the sender whenever requests are received, increasing computation overhead and delay. Finally, a peer needs to find proper peers to retrieve each vertex, which is expensive since the number of vertices is huge in a large progressive mesh.

To address these drawbacks, we adapted the receiver-driven protocol to the concept of chunk, commonly used in P2P systems. Our chunk, however, is of much finer granularity, to allow a peer the flexibility of retrieving only the mesh data within the current visible region. Each chunk consists of multiple vertex splits, and each vertex split only belongs to one chunk. When a peer needs a vertex split, it finds the chunk that includes this vertex split and sends its chunk

ID to request it. Each chunk is requested only once to avoid duplicate requests. Once a chunk is received, all the vertex splits in this chunk are decoded and processed.

This method sacrifices some flexibility in choosing vertices, but has several advantages. First, it significantly reduces the up-link bandwidth requirement since now only one chunk ID is needed to request a set of vertices. Second, the cost of searching for peers to retrieve data is also reduced. Third, grouping vertex splits into chunks can be done offline at the server, so no computation cost and time are needed by peers for online packetization.

The question now is how a peer knows which chunk to request when it decides to refine a certain part of a mesh. One naive solution is to store the chunk ID together with vertex splits, but this method adds relatively expensive overhead to each vertex split. For example, the chunk ID usually accounts for 2 bytes, while each encoded vertex split accounts for only 3-4 bytes, so the overhead is more than 50%. Another method we considered is to organize the mesh into many bounding boxes with the same size, and the vertex splits of the vertices inside a box belongs to a chunk. The chunk ID can thus be deduced from the vertex coordinates. Nonetheless, this method still needs extra information to associate the chunks with the bounding box. Moreover, since a vertex in a bounding box might have ancestors in other bounding boxes, this method increases dependencies among chunks, which might delay the decoding of some received vertices until other chunks are received.

In our solution, we define chunks based on vertex dependencies as follows. First, we simplified the mesh to s vertices. For convenience, we choose s to be the power of 2, i.e., $s = 2^i$. Then we split these vertices up to w levels and use the generated mesh as the base mesh. During the split, each of the original s vertices is split to 2^w vertices. Therefore, the base mesh has $s \times 2^w = 2^{i+w}$ vertices. We divide these vertices into s chunks and let the vertices split from a common ancestor to be in the same chunk. For each of these vertices, we also add its descendants up to d levels. Thus, a chunk comprises of 2^w subtrees of vertices, each of height d. The vertices in a chunk satisfy the following conditions: (i) their vertex IDs (binary form) have a common i-bit prefix; (ii) the bit length of their vertex IDs is in $[i+1, i+d]$.

Therefore, s chunks are available to be requested when only the base mesh is received, and each chunk has $2^w \times (2^d - 1)$ vertex splits. When a chunk is decoded, 2^{w+d} vertices will be generated.

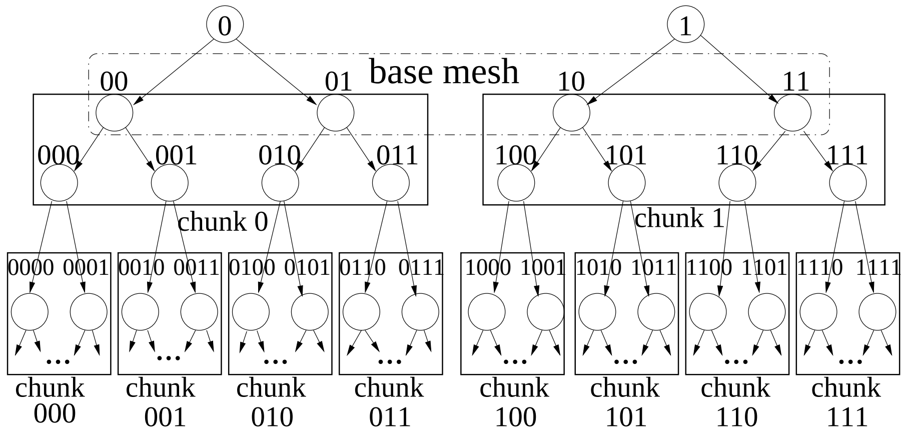


Figure 6.2: An example of hierarchical chunk structure. Here $w = 1$, $d = 2$ and $i = 1$.

We generate 2^d chunks from these 2^{w+d} vertices by putting the vertices which only differ in the last w bits into a chunk. These vertices are the roots of the subtrees in the chunk. Then, again for each vertex, we put its descendants up to d levels in the same chunk. Therefore, we now have a chunk hierarchy with 2^w root chunks and each chunk has 2^d children chunks. We call this chunk the *parent* chunk of these 2^d children.

All vertices in a chunk have a common ancestor, so we assign the vertex ID of this common ancestor as the chunk ID of this chunk (e.g. in Figure 6.2, the vertices in a chunk '000' are all children of vertex '000'). In other words, the chunk ID is the longest common prefix of the binary form of the vertex IDs of its members.

Figure 6.2 is a simple example with $i = 1$, $w = 1$, and $d = 2$. The base mesh has 4 vertices: 00, 01, 10 and 11. After the base mesh is received, two chunks 0 and 1 become available. The vertex splits for vertices 00 and 01, as well as vertices from 000 to 011 are inside the chunk 0. When chunk 0 is decoded, eight new vertices from 0000 to 0111 are generated. They become the root vertices of four new chunks from chunk 000 to chunk 011.

The main advantage of hierarchical chunk structure is that it fits the dependencies among the vertex splits well. First, such packetization does not increase the dependencies among chunks because vertices in a chunk are independent of any vertex splits except those inside the ancestor chunks. Second, both the dependency among chunk IDs and the relation between chunk ID and vertex ID are implicitly coded with the vertex ID. If the vertex ID of the root vertex is I_v, then the chunk ID is $I_c = (I_v >> d)$, and the ID of the parent chunk is $I_p = (I_c >> d) = (I_v >>$

$2d$). Since the vertex ID itself can be deduced as introduced in Chapter 4, we implement this hierarchical system without any extra cost.

Moreover, since the vertices in a packet are all children of a common ancestor, they are connected and are typically close to each other. Therefore, if one of the root vertices is needed by a peer, other root vertices are likely needed as well. Hence, it is reasonable to put them into one chunk.

With this packetization, a peer that needs some vertex splits, can deduce the IDs of the chunks to request from the vertex IDs. Some invisible vertex splits may be received, but because vertices inside a chunk are nearby each other, they may be visible soon anyway when this peer changes viewpoint.

We now describe how to determine the values of i, w and d. Suppose a chunk can carry about n vertex splits after compression. Therefore we have

$$2^w \times (2^d - 1) < n$$

Choosing a large w can improve the quality more quickly since typically the vertex splits in low levels contribute more to the mesh quality, but it increases the possibility of including many invisible vertices. In our implementation, each chunk is the size of one packet, although we can easily generalize to larger chunks. Since about 300 vertices can fit into an IP packet (i.e. $n = 300$), We choose $w = 4$ and $d = 4$, so a total of 240 vertex splits are in one chunk (if the progressive mesh is completely balanced).

After deciding the value w, the value of i can be decided based on the size of base mesh. A certain number of vertices is needed in the base mesh to ensure a minimal quality, and we know that the base mesh has $i \times 2^w$ vertices. So we can choose a proper i value to satisfy the quality of base mesh. In our implementation, we choose $i = 1024$. Thus, the base mesh has 16382 vertices.

6.5 Content Discovery

We now consider how a peer can discover other peers that can provide a given chunk. We first consider distributed hash table (DHT) and gossip-based techniques, commonly used in file sharing and video streaming, and discuss why they are unsuitable for our application. We then describe a simple centralized scheme that works reasonably well, followed by a more complex scheme that exploits the hierarchical structure in the data, trading off the number of messages processed by a server with response time and control overhead.

DHT, a well-known technique for content discovery, hashes a chunk to a peer, which maintains a directory of peers storing the chunk. DHT, however, is expensive to maintain and causes huge control overhead under high churn rate. As mentioned, short session time (typically several minutes) causes high churn rate in mesh streaming system. Moreover, DHT generally takes $O(\log n)$ hops to lookup, too long for the application we have in mind. We therefore preclude DHT in our solution.

Another common solution is a gossip-based protocol, in which peers exchange a bit-vector to notify each other which chunk it is holding on to. This approach is very flexible, but it does not fit well in P2P mesh streaming, either. The bit-vector is large due to small chunk size in our system, causing high control overhead. Second, due to short session times, much bandwidth is wasted in exchanging obsolete bit-vectors.

Compared to using DHT and gossip-based methods, maintaining a centralized lookup directory at the server has two advantages. First, the lookup is only a single hop. Further, the server can monitor peers' states and detect peer failures more quickly than in decentralized P2P systems, reducing the influence of churn. For these reasons, we study a P2P mesh streaming system with centralized lookup and evaluate its performance.

6.5.1 P2P With Centralized Lookup

We first consider one of the simplest ways of supporting content discovery, using a central lookup server that maintains the information of providers for each chunk of the mesh. The server maintains a list of peers who have downloaded each chunk. A peer p who needs a chunk i contacts the server. The server chooses a peer who has chunk i as provider and informs p (see Figure 6.3 (a)). Peer p informs the server after downloading chunk i, and is then added to the list of peers with chunk i. The server becomes the provider for a chunk i when there is no provider available for i or when p fails to receive a chunk from the given provider (e.g., if the provider has left or failed) (see Figure 6.3 (b)).

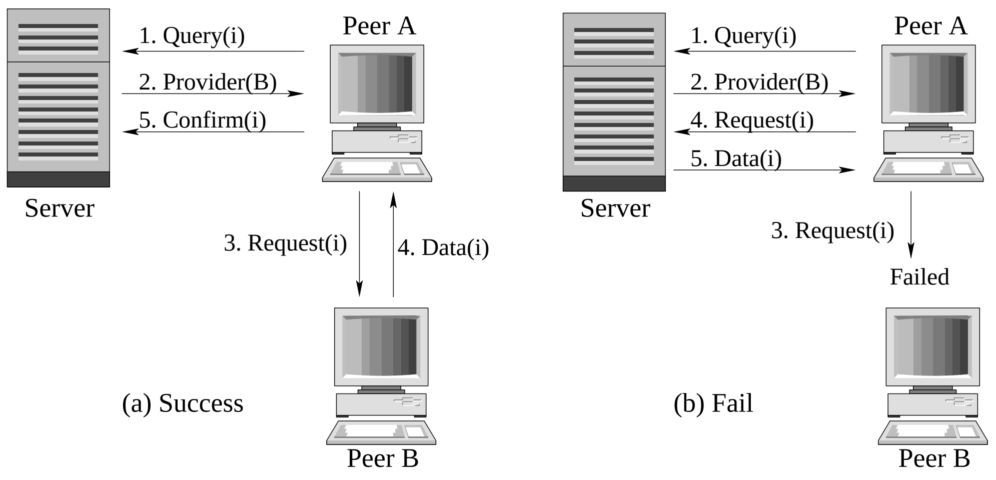


Figure 6.3: Centralized Lookup.

To keep the list of peers updated in an environment with high churn rate, the server monitors the list of providers. In our implementation, peers periodically (every 5 seconds in our implementation) send heart beat messages to the server, and the server "forgets" peers who have not been heard from after a certain period (15 seconds in our implementation).

The centralized lookup is suitable for a small or middle-scale P2P mesh streaming system. First, a query only costs two packets in terms of control overhead and one RTT in terms of delay. Second, states are maintained in one reliable node (the server), removing the need of synchronizing states among peers. Third, the server, maintaining all states, can employ better algorithms to determine the provider to improve the performance. For example, topology-aware provider selection can reduce the response time and inter-ISP traffic; load- and capability-aware provider selection can balance the load among peers and improve fairness.

Nonetheless, the centralized lookup is not scalable to a large number of peers, as provider monitoring and selection incur server overhead. Every request still causes a small packet to be sent from the server. We only decrease the size of outgoing packets, not the quantity. When the cost in handling requests is expensive (e.g. when topology-aware provider selection and load balancing is considered), the CPU becomes the bottleneck. For example, for a server with 100Mbps outgoing bandwidth and four 2GHz CPUs, assuming the outgoing packet size is 64 bytes, then the transmission delay is only $4.92\mu s$. If handling a request requires more than 40000 clock cycles, CPU becomes the bottleneck rather than the network bandwidth. Therefore, to further increase the scalability, we have to reduce the number of requests and the cost in handling requests.

To address this weakness, we propose a hierarchical P2P lookup approach, retaining most advantages of centralized lookup without introducing much overhead. We introduce this approach next.

6.5.2 Hierarchical P2P Lookup

In our hierarchical P2P lookup approach, a set of peers, called a *group*, is associated with each chunk of the mesh. A group G_i contains only peers that have already received chunk i. Each group G_i has a *leader*, denoted l_i, which acts similarly to the server in the P2P mesh streaming system with the centralized P2P lookup we introduced above. The members act as the providers for the chunk and are monitored by the leader. Since the group and the chunk have a one-to-one mapping, the leader of group G_i is also called the leader of chunk i.

Assume for now that a peer knows the group leader l_i given i (we will show how this is done later). When a peer p needs chunk i, it contacts l_i to join group G_i. The group leader l_i then selects a provider from the members of G_i and informs p of the identity of this provider. The peer p then contacts the provider for the chunk. Once p receives the requested chunk from the provider, it informs l_i and becomes a member of G_i.

The group leader acts as a provider when no other member exists. But, unlike the centralized lookup approach, if peer p fails to receive chunk i from a provider, it requests the chunk from the server rather than the group leader. The rationale here is that the server is more reliable than a group leader, so the response time can be guaranteed.

Note that one peer can belong to multiple groups. But a peer should not lead multiple groups simultaneously, (i) to ensure the peer has enough capacity to serve requests for one chunk, and (ii) to avoid a single peer failure affecting multiple groups.

Because a leader is also a peer, neither reliable nor having a well-known, fixed, address, two questions need to be answered: (i) how each peer knows the leader of a given group; (ii) how to deal with leader failure. We explain our solutions in the following sections.

Leader Hierarchy

Leaders inherit the hierarchical structure of chunks, due to the one-to-one mapping between groups and chunks. The leaders form a tree, mirroring the structure of chunks. We use the

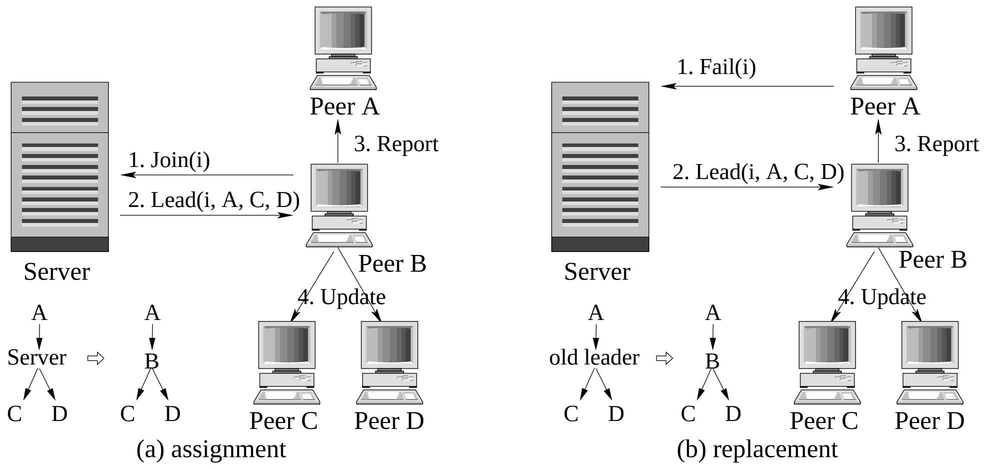


Figure 6.4: (a) Peer B is assigned a leader. (b) An old leader is replaced by peer B.

common terms leaf, root, parent, and child to describe leaders and their relationship in a self-explanatory way. Further, we consider the server as the parent of all root leaders.

Due to the dependency among the chunks, children of a chunk i are only useful after i is received. Hence, it is natural for a leader to supply the information of its children to its members. When a peer joins the group G_i, in addition to the information of provider s_i, the leader l_i also returns the information of its children. By exploiting the hierarchical property of the chunks, the leader information is obtained progressively without any query.

Leader Assignment and Replacement

In this section, we explain how leaders are assigned and how a failed leader is replaced. Initially, no peer exists in the system, and the server behaves as the default leader of all chunks. When a non-leader peer p joins a group G_i led by the server, the server assigns p as the leader of G_i. The server will not assign a peer as a leader if this peer is leading another group. When there are many candidates, how the server selects leaders is an interesting topic for further research. In our current implementation, the server only chooses peers with enough uploading bandwidth (larger than 512Kbps) as leaders.

Since leader assignments and replacements are done by the server, the server knows the current leader hierarchy. When assigning a peer p as the leader of group G_i, the server also includes information about p's parent and children in the tree. The new leader then notifies its parent and all its children of its new role.

To maintain the tree of leaders, a parent monitors all its children via heart beat messages. Failure of a child is reported by the parent to the server, which then chooses one non-leader peer from the peers who newly joined the system (see Figure 6.4(b)). The server maintains a list of recent, non-leader peers who query the server for root leaders initially. If no such peer is available, the server becomes the leader itself.

The leader replacement information is disseminated to the parent and children of the leader when the new leader advertised its role. The parent leader in turn notifies its own members of this leader change. So a member in the parent group of G_i will always know the updated leader of G_i.

It is possible that a peer may contact a failed leader and experience timeout. If a peer receives new leader updates from the server, it contacts the new leader; otherwise it requests the chunk from the server and reports the failure.

Reducing Monitoring Overhead

A peer becomes a member of group i when it receives chunk i. For a popular chunk i (such as the root), many members may exist. Monitoring and maintaining a large number of members is expensive for the leader. Furthermore, after receiving many chunks, a peer belongs to many groups, leading to high control overhead. This problem can be easily solved by restricting the group size and the number of groups a peer belongs to.

We reduce the number of groups a peer belongs to by the following rule: if a peer p obtained two chunks i and j, and i is an ancestor of j, then p only belongs to G_j.

By following the above rule, as a peer progressively obtains more and more chunks, it joins groups that are lower down the hierarchy and leaves the higher groups. Note that a peer leaves a group as long as it obtains *any* of the children chunks, not necessarily all. At a later time, a peer may want to download a child chunk i of a group it has left (perhaps due to a change of view point). But, since a peer is no longer in the parent group of the chunk i, it may not have up-to-date information about the leader of i.

To avoid this, a group leader propagates updates about its children down to its sub-tree. As a result, a peer always knows its leader, the leader's ancestors, and the immediate children of

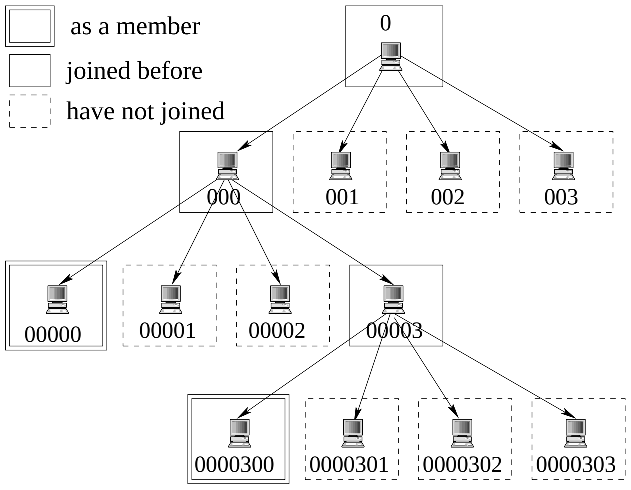


Figure 6.5: This peer will register in Group 00000 for leader updates from group 0 and 000, and it will register in Group 0000300 for leader updates from group 00003.

the leader's ancestors. Knowing the children of the leader's ancestors allows a peer to download any children chunk through the leaders.

This solution raises another problem. If a peer belongs to multiple descendants of a group G_i, it may receive duplicate leader updates, leading to unnecessary messages. Such duplicates can be removed by simply having this peer to explicitly register it to the leader closest (in terms of hops count in the leader hierarchy) to l_i for leader updates of G_i. This peer will not receive leader updates of G_i from other leaders since leaders only send its members the messages they have registered.

For example, as shown in Figure 6.5, a peer is a member in both group 00000 and 0000300. It has already received chunk 0, 000, and 00003 before, but has left those groups. Since it may join group 001, 002, and 003 in the future, it needs the leader updates from group 0. Group 00000 and 0000300 are both descendant groups of group 0, but 00000 is closer to group 0, so this peer registers to the leader of group 00000 for updates from group 0. Similarly, it registers to the leader of group 00000 for updates from group 000 and registers to the leader of group 0000300 for updates from group 00003. This method ensures that the leader updates will be received with the minimum delay and without duplication.

Note that only non-leader members may leave a group. To keep the leader hierarchy stable, the leader always stays in the group until it fails or the server assigns a new leader for this group.

Discussion

In the P2P system with hierarchical P2P lookup approach, each group behaves like a small P2P system with centralized lookup approach. First, most provider selection algorithms that can be used in a centralized lookup approach, such as topology-aware and load balancing, can also be used in the hierarchical P2P lookup approach. Second, providers are also monitored, so the failure rate of requests is low even with high churn rate. Third, query for a provider can be finished within 1 RTT in most cases. In short, most of the advantages of the centralized lookup approach are retained.

Compared with the centralized lookup approach, the management overhead of the server in the hierarchical P2P lookup approach is significantly reduced. In the stable stage, most group leaders are peers, so the server only monitors the leaders of the top chunks instead of all the peers in the centralized P2P system. In addition, in the hierarchical P2P lookup approach, the server is also responsible for replacing the failed leaders, and thus maintains the leader hierarchy at a small overhead. First, the number of leaders only relates to the number of chunks and thus is independent of the number of peers. Second, the rate of replacing leaders only relates to the leaving rate of the peers, which is typically small compared to the total number of peers. Thus, the management overhead is significantly smaller and the server is more scalable to a large number of peers compared to the centralized lookup approach.

The hierarchical P2P lookup approach, however, increases the control overhead and response time. It may also cause more chunks to be provided by the server. First, more control messages are needed to maintain the leader hierarchy, to propagate leader updates down to sub-trees, and to exchange heart beat messages between parent and children. Second, average response time increases since leaders may fail or leave. A peer who fails to contact a leader needs an additional RTT (to server) before receiving a chunk. Third, leader failure causes the server to provide more chunks. Another reason is that some valid providers may not be used after they leave a group, which will not happen in the centralized lookup approach.

In the next section, we further evaluate the centralized and hierarchical P2P lookup approaches by simulations.

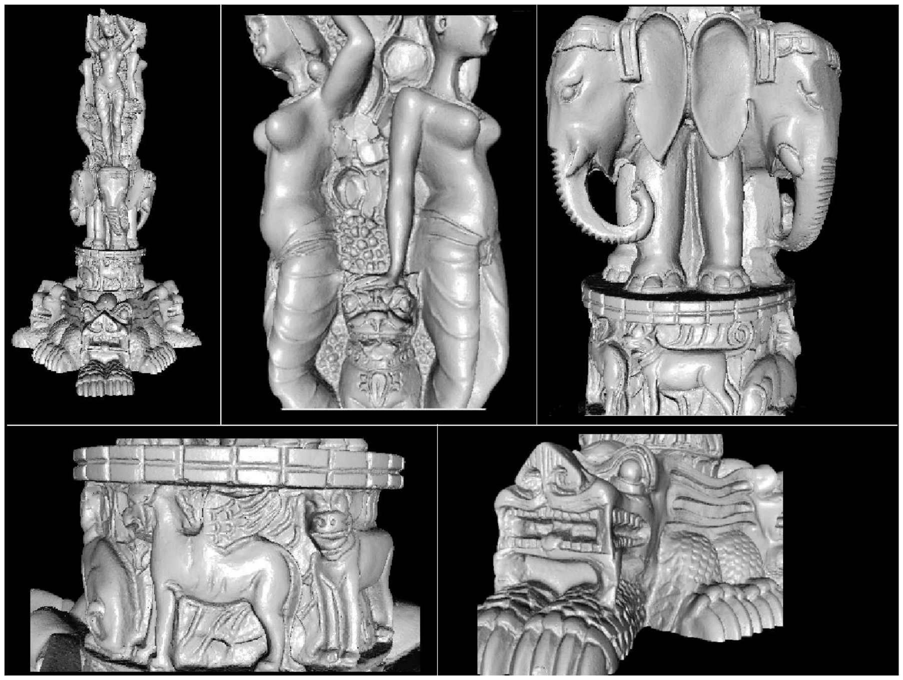


Figure 6.6: The Thai Statue. The top left is the whole mesh, and the rest is the closeup of some parts of it. Original mesh courtesy of Stanford Computer Graphics Laboratory.

6.6 Experimental Results

In this section, we present the experimental results to evaluate the performance of the two lookup approaches we studied: centralized lookup and hierarchical P2P lookup.

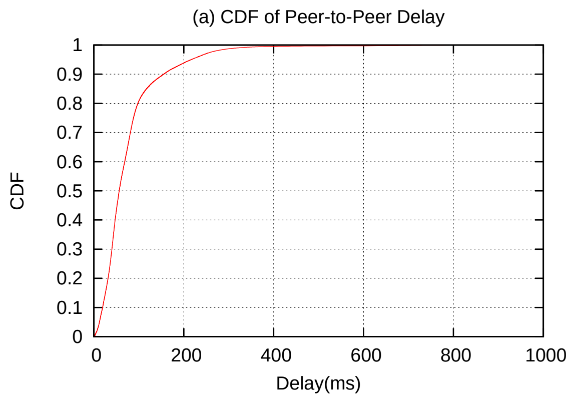
6.6.1 Experiment Setup

We developed two systems to support the experiments: (i) a receiver-driven client and server implementing view-dependent streaming of progressive meshes used to collect and generate traces; and (ii) a simulator that replays the traces generated and simulates large-scale P2P mesh streaming.

The client-server system is programmed in C++ based on OpenGL and OpenMesh[2]. In our experiments, we use the Thai Statue from Stanford Computer Graphics Laboratory, which has 5 millions vertices and takes up 22.5 MB after being compressed with our encoding method. The mesh is packetized into 147897 chunks following the approach introduced in Section 6.4. We choose this mesh since details exist all over the statue, so users may have interest in different positions and levels of details (see Figure 6.6).

[2]http://www.openmesh.org

To simulate many users, we generated 6000 synthetic traces following the model introduced in Section 5.4. We replay these generated traces to obtain the sequence of chunk requests and use the latter in the simulation.



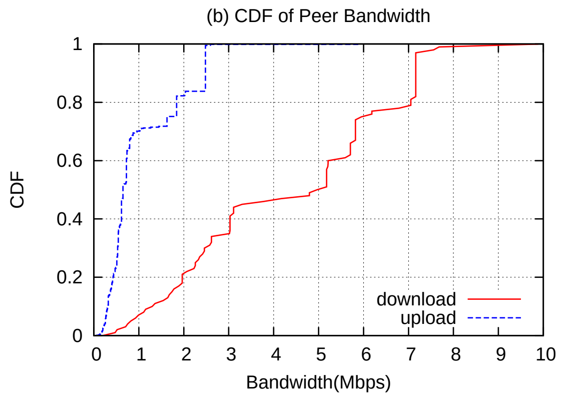


Figure 6.7: CDF of peer-to-peer delay (mean = 75.8ms) and peer bandwidth (download mean = 4292.8Kbps, upload mean = 1023.0Kbps) used in our simulation.

The P2P simulator is a discrete-event simulator based on OMNeT++. In our simulator, peers follow a Poisson arrival model. Each peer randomly selects a trace from 4000 random traces we generated and leaves the system at the end of the trace. Our experiment lasts for 320 seconds, and the period from $300s$ to $320s$ is the stable stage (approximately). We choose the arrival rate (λ) as 20, 40, 60, and 80. During the stable stage, the number of online peers is around 1974, 4030, 6014, and 7974, respectively.

The end-to-end delay between two peers (including server to peers) is taken from the data collected from the Meridian [13] project. The original data is a 2500×2500 matrix recording the pair-wise delay between 2500 DNS servers. We assign each peer and the server a random

DNS server and assume that the end-to-end delay of two peers equals the delay between their DNS servers (see Figure 6.7(a) for the CDF of end-to-end delay).

The downloading and uploading bandwidth of peers are randomly selected from the data reported by DSLReports.com [3], which records and updates daily the access speed of users over the world. We use 21,206 samples, collected on 14 April 2009. Figure 6.7(b) shows the CDF of the downlink and uplink bandwidth.

In the following sections, we evaluate our P2P design in terms of server overhead, incoming message rate of the server, control overhead, and average response time.

6.6.2 Server Overhead

In this section, we measure the server overhead in three forms. First, we measure the outgoing data rate of the server. Second, we measure the ratio of the server's outgoing data rate to the total chunk size received by the peers. It is a *relative server overhead* compared to the client-server model based system. Third, to evaluate how busy a server is in handling incoming messages, we measure the incoming message rate of the server to indicate the server overhead in handling incoming requests.

The outgoing data rate is shown in Figures 6.8(a) and (b). Figures 6.8(a) shows that the server outgoing data rate in the client-server model exceeds 1Gbps when λ exceeds 40. It indicates that the client-server model cannot scale well without increasing the number of servers and outgoing bandwidth, so we will not consider the client-server model in further comparisons.

Figures 6.8(c) and (d) show the relative server overhead. The relative server overhead of the hierarchical P2P lookup is slightly larger than that of the centralized lookup in our implementation. The main reason is that more chunks are provided by the server in hierarchical P2P lookup than that in centralized lookup, as we will discuss in the section about response time.

The server incoming message rate of the two designs is shown in Figures 6.8(e) and (f), where we can see that the hierarchical P2P lookup reduces the number of incoming messages by around 80%. Moreover, the incoming message rate increases more slowly with λ in hierarchical P2P lookup, indicating that the CPU overhead of the server (to handle the messages) also increases more slowly.

[3]http://www.dslreports.com/archive

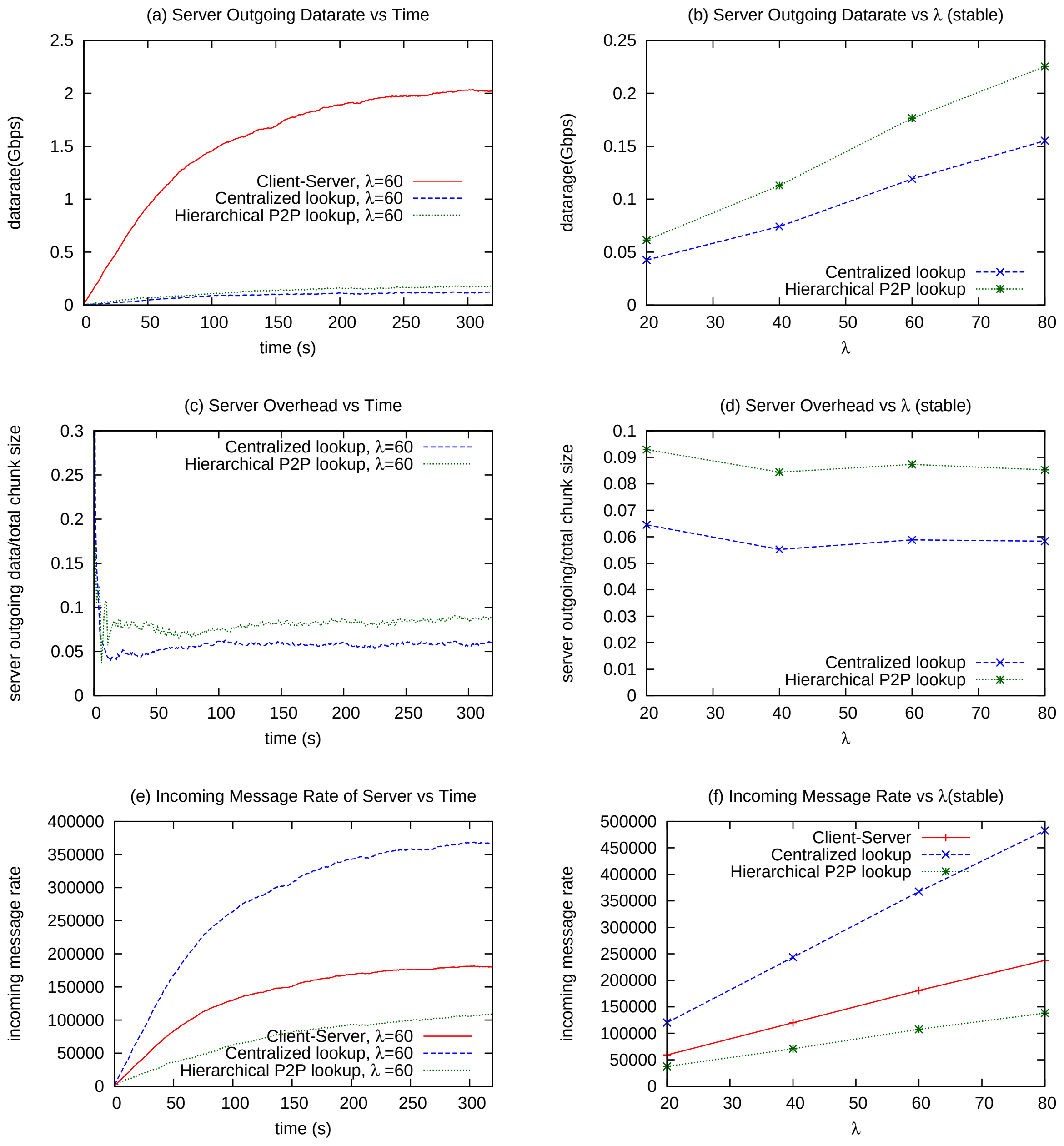


Figure 6.8: Comparison between centralized lookup approach and hierarchical P2P lookup approach. The left column indicates how the results change with time, and the right column how the results change with the arriving rate of peers. The value in stable stage is averaged from $t = 300s$ to $t = 320s$.

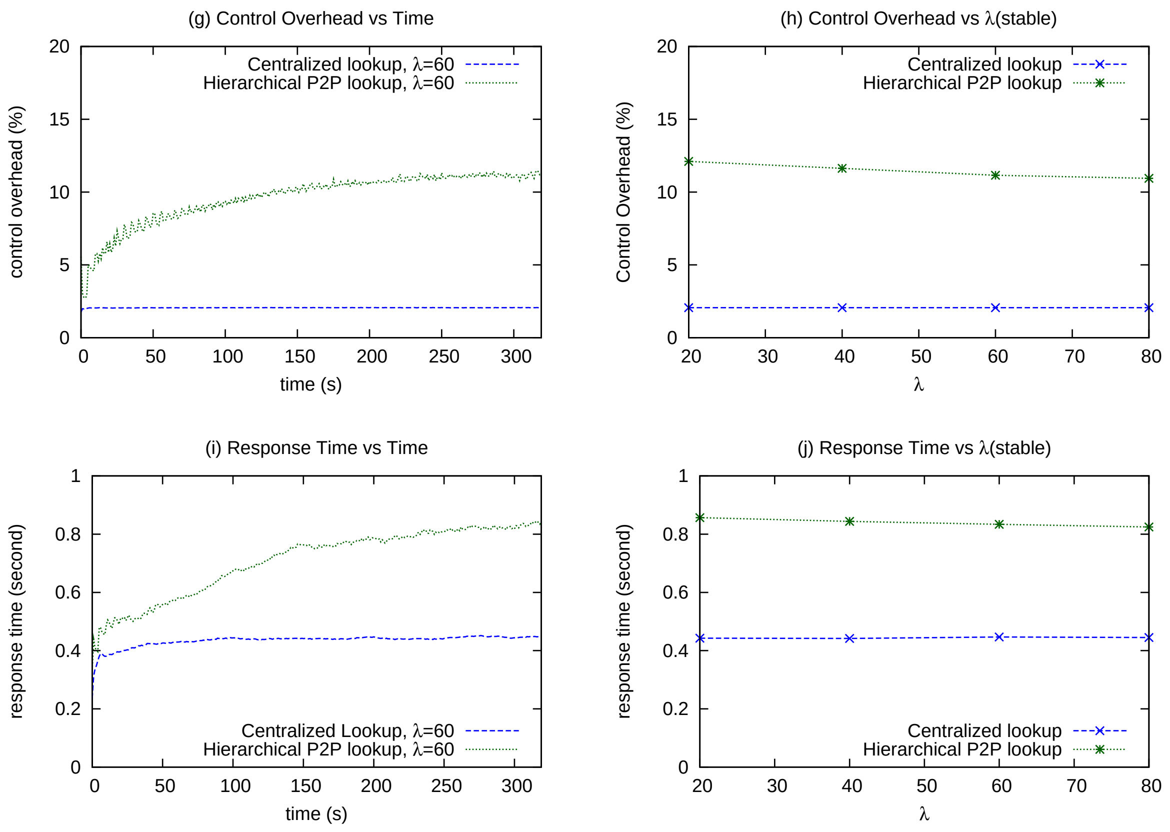


Figure 6.9: Comparison between centralized lookup approach and hierarchical P2P lookup approach. The left column indicates how the results change with time, and the right column how the results change with the arriving rate of peers. The value in stable stage is averaged from $t = 300s$ to $t = 320s$.

Note that in these two designs, the server responses to incoming requests are different. The responses are leader information (for failed *join*s) and chunks (for failed *request*s) in hierarchical P2P lookup approach, but are mainly provider information in centralized approach. Retrieving a chunk or leader information is typically cheaper than deciding the best provider, especially when topology awareness and load balancing are considered. Hence hierarchical P2P lookup not only reduces the incoming message rate, but also reduces the handling overhead per request to the server.

6.6.3 Control Overhead

In this section we examine how much network bandwidth is used by control messages. We define control overhead as the ratio of the size of control messages to the size of chunks received by all peers during one second. Figures 6.9 (g) and (h) show that hierarchical P2P lookup has higher control overhead than centralized lookup. The higher overhead is caused by maintaining the leader hierarchy and propagating leader updates. Moreover, in hierarchical P2P lookup, peers may belong to multiple groups, leading to more messages used in monitoring. Nonetheless, the control overhead of both systems are within an acceptable range.

6.6.4 Response Time

Another important metric to evaluate the P2P mesh streaming system is the response time. The response time for each chunk request is obtained by subtracting the receiving time of the chunk by the sending time of the corresponding request. Since no strict order in receiving chunks is required in mesh streaming, the response time of a single request is not as important as that in audio and video streaming. Instead, we emphasize on the average response time of requests sent in a second, which is the average value of all the response times of chunk requests in a second.

Figures 6.9(i) and (j) show that the hierarchical P2P system has higher average response time. The higher response time is caused mainly by two reasons. First, unlike providers, only one leader exists for one chunk. Leader failures cause many peers to spend one more round trip to obtain chunks. It is possible to use multiple leaders in a group in the future to reduce this effect. Second, a leader in hierarchical P2P lookup approach does not have the information of all the providers as the server in centralized lookup, and congestion may happen due to limited supply ability, causing request failures and higher response time.

The response time of the hierarchical P2P approach, however, is still under 1 second, which according to our experience, does not significantly affect users' experience. When a user interacts with the mesh (e.g., rotate), the renderer responds almost immediately. The response time of chunk requests only affects how fast the mesh quality improves after users change their view points.

6.6.5 Summary

The results above indicate that both designs work well under an environment with heterogeneous peers, asymmetric bandwidth, and high churn rate. Compared to a client-server design, the server outgoing bandwidth is reduced by more than 90%. Hierarchical P2P lookup also reduces 60% of incoming messages to the server, but generates only around 10% of control overhead when the number of peers is large. The average response time of both systems is below a second and does not affect the user experience due to the progressive rendering nature of the mesh. Further, as the number of peers increases, the control overhead and response time remain relatively stable.

6.7 Conclusion

This chapter investigates the problem of P2P view-dependent, progressive mesh streaming, and studied two important components of the problem - chunking and content discovery. We find that in P2P mesh streaming, peers need to keep finding new chunk providers, increasing the control overhead. Furthermore, the short session length of peers increases the churn rate. We considered these unique characteristics of mesh streaming and explored two content discovery schemes. We found that centralized lookup works well with these challenges. To further reduce the CPU overhead of the server, a hierarchical P2P lookup approach can be used to move the lookup service to selected peers.

Chapter 7

Conclusion

This thesis concentrates on streaming of high-resolution progressive meshes over the network. The objective is to improve the quality of rendered image in the receiver side as quickly as possible, or in other words, to improve the quality curve of reconstructed mesh in the receiver side.

This thesis contributes to this objective in three aspects. First, we quantified and minimized the negative effect of dependency among the vertex splits when packet losses happen during transmission. Second, we proposed a more scalable way to implement view-dependent streaming system, which can significantly improve the rendering quality curve by considering each user's viewpoint. Finally, with the help of the receiver-driven protocol we proposed, we developed a prototype of P2P view-dependent progressive mesh streaming system so that a receiver can request data from many peers to improve the quality of the reconstructed mesh.

In this chapter, we briefly review the main contributions of this thesis and point out the possible future work, including both extensions of existing work and new research problems.

7.1 Contributions and Extensions

7.1.1 An Analytical Model to Quantify the Effect of Dependency and Packet Loss

When progressive meshes are streamed over lossy networks, a packet loss affects not only the vertex splits inside this packet, but also all the following vertex splits with ancestors in this packet. Hence, the dependency is one of the important factors to be considered in deciding the sending order when progressive meshes are streamed over the network. In this thesis, an analytical model was developed to estimate the quality curves generated by various sending orders, with the dependency among vertex splits and the network conditions as the input. Many experiments were carried out to verify the accuracy of this analytical model, and different network traces and progressive meshes were used in these experiments to avoid bias. According to the experimental results, the predicted decoding time of vertex splits using the analytical model was reasonably close to the measured one. As a result, this analytical model could be used to efficiently predict the quality of the received mesh before transmission.

In the experiments, we observed that the effect of dependency is only significant in the short period at the beginning of the transmission, and the length of this period increases when the round trip time or the packet loss rate increases. This observation is consistent with the results deduced from the analytical model. Here we give an intuitive explanation. Each lost packet will be retransmitted as soon as the packet loss is detected, so the effect of a packet loss is restricted to a certain period. During the transmission, the effect of lost packets decreases because (i) the contribution of a packet decreases; (ii) the quality of the reconstructed mesh increases. Consequently, the relative value of a packet, represented as the ratio of its contribution to the quality of the reconstructed mesh, decreases. As a result, the dependency among the vertex splits only needs to be considered in applications requiring high interactivity, because these applications require low initial latency. For these applications, a greedy method was proposed in this thesis to generate an adaptive sending strategy to reduce the initial latency.

Some extensions could be done to improve the analytical model introduced in this thesis. First, the precision of the model could be improved by considering bursty packet losses. The analytical model in this thesis assumes that the packet losses are independent of each other, so a network trace with many bursty packet losses may cause considerable error in the prediction, as we already showed in the experimental results in Chapter 3. Gilbert model of packet loss may be used to obtain more accurate results, but using Gilbert model may significantly increase the complexity of the analytical model. Considering bursty packet losses happen less commonly now in the Internet after Random Early Detection (RED) [31] was widely used in routers, It needs to be investigated first whether the benefit of applying Gilbert model justifies the increasing complexity.

Second, our model is general enough for any partially ordered data as long as it can be represented as a DAG. For example, this model was successfully applied in streaming of 3D plants [59, 60]. Many other media types also satisfy this requirement. For example, in point-based representations of 3D objects, the dependency could be represented as trees, and a tree is a simple DAG. In Layered Video Streaming (e.g. H264-SVC), the dependency in one GOP is also a tree. Hence, the model in this thesis could also be extended to support modeling streaming of these media types. Moreover,a tree is simpler than a general DAG in that each node has only one direct parent. As a result, it is possible to compute the decoding time in a recursive way, which is easier than the method introduced in Chapter 3. If we know the distribution of the decoding time of vertex v, D_v, then the decoding time of its child c can be represented as $min(R_c, D_v)$, where R_c is the receiving time of vertex split c.

Third, the quality of the received mesh in the first several seconds can be further improved by introducing FEC (Forward Error Correction) packets, so that the packet loss may be recovered before the retransmission. Introducing FEC packets, however, is not free of costs because it will delay the sending time of following packets. Hence, a trade-off between sending more FEC packets to increase the recovery probability and sending less FEC packets to send following vertex splits earlier should be made to optimize the quality curve. Our model may help in finding the proper trade-off because the quality curve can be estimated efficiently for various number of FEC packets.

7.1.2 Receiver-Driven View-Dependent Mesh Streaming Protocol

View-dependent streaming significantly improves the quality curve by considering each user's viewpoint. Current implementations, however, cannot be easily scaled to a large number of receivers as the sender is stateful. To address this weakness, a receiver-driven view-dependent streaming system, in which the sender is stateless, was proposed in this thesis. In this system, the visibility decision is moved from the sender side to the receiver side, and the server now only sends back the data on-demand. Hence, a significant advantage of the receiver-driven approach is that now both the caching proxy and the P2P technique can be applied to increase the scalability of the system as the sender is stateless.

One main challenge of the receiver-driven protocol is to enable the receiver to determine the visibility and visual contribution of the vertex splits before receiving them. In this thesis, the screen area of a vertex split was used to estimate its visibility and visual contribution. A method to detect the silhouette was also proposed and high priority could be assigned to the silhouette when the profile of the mesh is important. Experimental results showed that these methods both work well in improving the quality curve.

Further research could investigate into more sophisticated methods to estimate the visual importance of vertex splits. For example, we could consider the roughness, the transient level, or the semantic meaning of different parts of the mesh in estimating their visual importance.

In this thesis, silhouette is simply detected by looking for pixels being neighbor to background pixels. Self silhouette cannot be detected by this method. More sophisticated silhouette detection algorithms can be used to address this problem. For example, the depth of each pixel or the variation of face normal values could be used in detecting the silhouette. The efficiency of these methods, however, should be considered to prevent the silhouette detection from being the bottleneck of the real-time rendering.

Finally, finding a proper objective view-dependent quality metric remains a challenging problem. In our experiments, the PSNR value of rendered images was used in comparing the quality of reconstructed meshes. It sometimes did not comply well with the human perception. For example, an image with a block error in a small area may have the same PSNR value as an image with error pixels scattered all over the image. The former, however, has worse perception because a block error is easier to be detected and distracts the viewer's attention. Some other objective quality metric, such as SSIM, could be further investigated to address this problem.

In the receiver-driven approach proposed in this thesis, the visibility decision of the receiver may not be completely accurate since the receiver can only estimate the visibility based on the partially received mesh. Visible vertices may not be displayed if its parent vertex has no visible neighbor faces. According to the current results, however, this kind of error is negligible in most cases. Moreover, we found that this error can be significantly reduced by force splitting the neighbors of vertices in the silhouette. For applications that tolerate no error, however, further studies are required to completely remove this error.

7.1.3 P2P View-Dependent Progressive Mesh Streaming

To increase the scalability of the view-dependent mesh-streaming systems, a P2P mesh streaming system was proposed in this thesis based on our receiver-driven approach. Two main problems in implementing a P2P view-dependent mesh streaming system were addressed.

The first problem is how to group vertex splits into chunks. Chunking in P2P mesh streaming system cannot follow a fixed order as the time order in P2P video streaming because of the variety in user interests. In this thesis, a chunking method following the vertex hierarchy is proposed and implemented. With this method, which chunk a vertex split belongs to can be implicitly known, and all vertex splits in a chunk are close to each other. Moreover, the dependency among chunks are minimized such that each chunk has only one direct parent.

The second problem is how to design the content discovery system when peers have to frequently looking for new providers in a P2P view-dependent progressive mesh streaming system. Two methods, central lookup and hierarchical P2P lookup system, are proposed to target networks with different scales. To verify the effect of these two methods, we do simulations using practical parameters obtained from the Internet. According to the simulation results, the server overhead can be reduced by 90% with the cost of slightly longer response time (less than one second) and larger control overhead (around 10%).

The control overhead is around 10% in a hierarchical P2P lookup system, and the larger control overhead is mainly caused by the leader informations exchanged among leaders and between leaders and members. How frequently these informations are exchanged depends on the updating interval, a preset parameter. Using a larger updating interval could reduce the control overhead. The response time, however, may increase due to more request failures caused by out-date informations. Hence, further research could investigate the effect of updating interval and try to find a proper updating interval to trade off between control overhead and response time.

The response time, which is less than one second in both systems, is acceptable in many applications without strict real time requirement. For those application requiring strict low latency, such as 3D games, pre-fetching can help to reduce the response time. To enable effective pre-fetching, the user's next action should be predicted with high accuracy. The preliminary work we have done on studying user behaviors has already been introduced in Chapter 5. The confidence level of our results, however, is greatly compromised by limited number of samples. In the future, if we could develop a practical application, such as an online shop or an online museum, and put it online, we may collect a large number of real traces, with which we could better understand the pattern of user behaviors and have a higher confidence in predicting user actions.

The server overhead of the hierarchical P2P lookup system is slightly higher than the centralized lookup system, mainly due to the leader failures. Once a leader fails, many requests will go to the server for help. The server has to behave as a provider temporarily because the provider list is lost with the failed leader together. To overcome this problem, a backup leader, which maintains a copy of the provider list, could be introduced for each chunk. Once a leader fails, the server can immediately promote the backup leader and the provider list is recovered. As a result, the chance that the server has to be a backup provider can be greatly reduced.

7.2 Other Future Work

Streaming of progressive meshes is still a relatively new area, where many interesting research problems exist. In the last part of this thesis, in addition to the extensions mentioned above, some selected new research problems are listed as future works. These research problems are categorized into three groups: coding (in the sender side), transmission (between the sender and the receiver), and rendering (in the receiver side).

7.2.1 Coding

The main objective of the coding of progressive meshes is to compress vertex splits more efficiently. Meanwhile, the flexibility in selectively requesting vertex splits cannot be sacrificed. In the implementation introduced in this thesis, each vertex split can be separately requested even after compression. In some cases, such as the P2P system introduced in Chapter 6, where the data is organized in the chunk level, this kind of vertex split level granularity is not necessary. Therefore, it is possible to exploit the correlation between vertex splits in the same chunk to further improve the compression efficiency. For example, coding the geometry data of vertex splits can be improved by predicting the coordinates of new vertices from the surrounding vertices and coding the error of the prediction with entropy coding algorithms.

Another approach to improve the geometry coding is to encode coordinates with progressive precision. Currently, the coordinate of a vertex has the highest precision immediately after it is generated. This method unnecessarily increases the data size streamed in the initial stage. As we showed in Chapter 4, in the initial stage, the geometry code of a vertex split costs more than 32 bits but it may cost as low as 16 bits later. This cost is not desired in some applications requiring low response time in the initial stage. To address this problem, it is possible to send coordinates with low precision first, and improve the precision by sending extra data later (e.g. when this vertex is to be split). Then the size of geometry data of a vertex split can be more even and the initial quality of reconstructed mesh could be improved faster.

Another interesting problem is how to generate the base mesh. First, it is useful to know what is the optimal size of the base mesh. The base mesh should be large enough to provide acceptable initial quality, but cannot be too large to avoid long downloading time. Furthermore, how to generate the base mesh is also important. A carefully generated base mesh will have better silhouette and reduce the error in estimating visibility by the receiver.

This thesis concentrates on coding one single large mesh without textures. Further studies could be done to enable coding textures and a scene with many meshes. The textures themselves can be compressed as images with mature algorithms. For example, the progressive JPEG, could be a good candidate. Two main problems, however, still remain to be solved. First, how to integrate the textures with the meshes together to support progressive streaming. Second, how to ensure the texture coordinates of each vertex to be consistent in different level of details.

To extend our system to a scene with multiple meshes could be implemented by adding one more level, including mesh level visibility detection and mesh level importance determination. I believe these directions lead to a large amount of work that themselves are enough for another Ph.D. thesis. Here, only some preliminary thoughts are listed. First, occlusion detection and Z-culling algorithm could be used to remove those invisible objects efficiently. Second, screen-area based method may also be applied to mesh level importance determination. Usually, the objects that are far from the viewer have lower importance. At the same time, the faces of these far objects have smaller screen-area. Therefore, screen-area based method already takes the distance of objects into consideration.

7.2.2 Transmission

End-to-end delay is another important factor to affect the user experience in a mesh streaming system. Many research problems relate to how to reduce the end-to-end delay. First, caching proxies that deployed close to the end users could significantly reduce the end-to-end delay. To effectively use the resource of a caching proxy, it is important to know which part of a mesh is most popular. We have obtained some interesting findings from the preliminary analysis of the real user traces introduced in Chapter 5. We found that in our limited number of traces and specific usage scenario, the locality exists in the data access pattern of users. Therefore, it is possible to store a small part of the mesh and satisfy a large number of requests. Moreover, our study shows that user actions are somewhat predictable. Hence, pre-fetching based on predicting users' next actions can also be used to significantly compensate the end-to-end delay.

For some applications, it is important to ensure the data is from the trusted sources without being tampered. As a result, authentication is required if caching proxies or P2P techniques are applied in these applications. To enable finding the corrupted data as soon as possible, the authentication should be done progressively. It is non-trivial to implement the progressive authentication in mesh streaming system because the data may be sent in many different orders and the sender does know the sending order in advance. How to embed the authentication information into progressive mesh then becomes an interesting and challenging research problem.

7.2.3 Rendering

In some cases, we found that the rendering of meshes in the local computer can be the bottleneck in a mesh streaming system, especially when the GPU is also used to decide the visibility of vertices. Improving the rendering efficiency of the progressive mesh during the streaming could be an interesting research problem. During the streaming, a progressive mesh with only partial modification is frequently re-rendered, so it is possible to exploit the similarity between two consecutive version of reconstructed meshes to improve the rendering speed. For example, we could represent some of the rendered details as the textures in the next version of the mesh to avoid redundant rendering. Moreover, it is interesting to see whether some rendered part in the frame buffer can be reused in the following frames as long as the viewpoint is not changed.

Moreover, remote rendering, in which the sender sends the rendered images directly to the receivers, is an effective way to enable devices without enough rendering power to join the mesh streaming system. An extended P2P systems, where some peers not only provide the raw vertex splits but also provide rendered images, could be developed to further enable the 3D streaming applications on resource constraint devices.

Appendix A

Related Publications

This thesis is based on several our published/accepted papers.

- Chapter 3 is based on our paper accepted in TOMCCAP [23], which itself is an extended version of our paper published in ACM Multimedia 2007 [22].
- Chapter 4 is extended from our paper published in NOSSDAV 2008 [21] by adding the evaluation of screen-based method based PSNR curves.
- Chapter 5 is partly based on our short paper published in ACM Multimedia 2009 [72], except that the synthetic trace generation is newly added and the prediction part has been rewritten.
- Chapter 6 is based on our full paper published in ACM Multimedia 2009 [20].
- The abstract of this thesis is presented at ACM Multimedia 2008 as a Doctoral Symposium Abstract [19].

Bibliography

[1] G. Al-Regib and Y. Altunbasak. An unequal error protection method for packet loss resilient 3D mesh transmission. In *Proceeding of the IEEE INFOCOM 2002*, pages 743–752, New York, NY, USA, June 2002.

[2] G. Al-Regib and Y. Altunbasak. 3TP: An application-layer protocol for streaming 3D models. *IEEE Transactions on Multimedia*, 7(6):1149–1156, December 2005.

[3] P. Alliez and M. Desbrun. Progressive compression for lossless transmission of triangle meshes. In *Proceedings of the SIGGRAPH 2001*, pages 195–202, Los Angeles, CA, USA, August 2001.

[4] P. Alliez and M. Desbrun. Progressive compression for lossless transmission of triangle meshes. In *Proceeding of the SIGGRAPH 2001*, pages 195–202, New York, NY, USA, August 2001.

[5] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 480–489. 2001.

[6] P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. In M. S. N.A. Dodgson, M.S. Floater, editor, *Advances in Multiresolution for Geometric Modelling*, pages 3–26. 2005.

[7] P. Alliez, N. Laurent, H. Sanson, and F. Schmitt. Efficient view-dependent refinement of 3D meshes using sqrt(3)-subdivision. *The Visual Computer*, 19(4):205–221, July 2003.

[8] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings of the ICME 2002*, pages 705–708, Lausanne, Switzerland, 2002.

[9] D. I. Azuma, D. N. Wood, B. Curless, T. Duchamp, D. H. Salesin, and W. Stuetzle. View-dependent refinement of multiresolution meshes with subdivision connectivity. In *Proceedings of the AFRIGRAPH 2003*, pages 69–78, Cape Town, South Africa, February 2003.

[10] C. L. Bajaj, V. Pascucci, and G. Zhuang. Progressive compressive and transmission of arbitrary triangular meshes. In *Proceedings of the IEEE Visualization 1999*, pages 307–316, Los Alamitos, CA, USA, October 1999.

[11] C. L. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *Proceedings of the DCC 1999*, pages 247–256, Snowbird, UT, USA, March 1999.

[12] P. Bao and D. Gourlay. A framework for remote rendering of 3-D scenes on limited mobile devices. *IEEE Transactions on Multimedia*, 8(2):382–389, 2006.

[13] A. S. Bernard Wong and E. G. Sirer. Meridian: A lightweight network location serive without virtual coordinates. In *proceedings of the SIGCOMM 2005*, pages 85–96, Philadelphia, PA, August 2005.

[14] J. M. Boyce and R. D. Gaglianello. Packet loss effects on mpeg video sent over the public internet. In *Proceedings of the ACM Multimedia 1998*, pages 181–190, Bristol, UK, September 1998.

[15] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proceedings of the SOSP 2003*, pages 298–313, Sagamore, NY, USA, October 2003.

[16] R. Cavagna, C. Bouville, and J. Royan. P2P network for very large virtual environment. In *Proceedings of the ACM VRST 2006*, pages 269–276, Limassol, Cyprus, November 2006.

[17] Z. Chen, J. F. Barnes, and B. Bodenheimer. Hybrid and forward error correction transmission techniques for unreliable transport of 3D geometry. *Multimedia Systems*, 10(3):230–244, 2005.

[18] I. Cheng, L. Ying, and A. Basu. Packet-loss modeling for perceptually optimized 3d transmission. *Advanced Multimedia*, 2007(1):11–11, 2007.

[19] W. Cheng. Streaming of 3d progressive meshes (doctoral symposium abstract). In *Proceedings of ACM MULTIMEDIA 2008*, Vancouver, Canada, October 2008.

[20] W. Cheng, D. Liu, and W. T. Ooi. Peer-assisted view-dependent progressive mesh streaming. In *Proceedings of ACM MULTIMEDIA 2009*, Beijing, China, October 2009.

[21] W. Cheng and W. T. Ooi. Receiver-driven view-dependent streaming of progressive mesh. In *Proceedings of the NOSSDAV 2008*, pages 9–14, Brauschweig, Germany, May 2008.

[22] W. Cheng, W. T. Ooi, S. Mondet, R. Grigoras, and G. Morin. An analytical model for progressive mesh streaming. In *Proceeding of the ACM MULTIMEDIA 2007*, pages 737–746, Augsberg, Germany, September 2007.

[23] W. Cheng, W. T. Ooi, S. Mondet, R. Grigoras, and G. Morin. Modeling progressive mesh streaming: Does data dependency matter? *accepted by ACM Transactions on Multimedia Computing, Communications, and Applications(TOMCCAP)*, 2009.

[24] M. M. Chow. Optimized geometry compression for real-time rendering. In *Proceedings of the IEEE Visualization 1997*, pages 347–354, Phoenix, AZ, USA, October 1997.

[25] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of the SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, USA, June 2000.

[26] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[27] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings of the IEEE Visualization 1999*, pages 67–72, San Francisco, CA, USA, October 1999.

[28] Y. Cui, B. Li, and K. Nahrstedt. oStream: asynchronous streaming multicast in application-layer overlay networks. *IEEE journal on selected areas in communications*, 22(1):91–106, January 2004.

[29] H. de Roos. The digital sculpture project. *Computer and Information Science*, 9(2), 2004.

[30] M. Deering. Geometry compression. In *Proceeding of the SIGGRAPH 1995*, pages 13–20, Los Angeles, CA, USA, August 1995.

[31] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, 1993.

[32] P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *Proceedings of the SIGGRAPH 2002*, pages 372–379, San Antoria, Texas, USA, July 2002.

[33] C. Gotsman, S. Gumhold, and L. Kobbelt. Simplification and compression of 3D meshes. In *Tutorials on Multiresolution in Geometric Modeling*. 2002.

[34] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *Proceedings of the SIGGRAPH 2002*, pages 355–361, San Antoria, Texas, USA, July 2002.

[35] Y. Gu and W. T. Ooi. Packetization of 3D progressive meshes for streaming over lossy networks. In *Proceedings of the ICCCN 2005*, pages 415–420, San Diego, CA, October 2005.

[36] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. In *Proceedings of the SIGGRAPH 1998*, pages 133–140, Orlando, FL, USA, July 1998.

[37] A. F. Harris(III) and R. Kravets. The design of a transport protocol for on-demand graphical rendering. In *Proceedings of the NOSSDAV 2002*, pages 43–49, Miami, FL, USA, May 2002.

[38] Y. He, B.-S. Chew, D. Wang, S. C. Hoi, and L.-P. Chau. Streaming 3d meshes using spectral geometry images. In *Proceeding os the ACM Multimedia 2009*, pages 431–440, Beijing, China, October 2009.

[39] H. Hoppe. Progressive meshes. In *Proceedings of the SIGGRAPH 1996*, pages 99–108, New Orleans, LA, USA, August 1996.

[40] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the SIGGRAPH 1997*, pages 189–198, Los Angeles, CA, USA, August 1997.

[41] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, Jan-Feb 1998.

[42] S.-Y. Hu, T.-H. Huang, S.-C. Chang, W.-L. Sung, J.-R. Jiang, and B.-Y. Chen. FLoD: A framework for peer-to-peer 3D streaming. In *Proceedings of the IEEE INFOCOM 2008*, pages 1373–1381, Phoenix, AZ, USA, April 2008.

[43] J. Jannotti, D. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O. Jr. Overcast: reliable multicasting with on overlay network. In *Proceedings of the OSDI 2000*, pages 1–14, San Diego, CA, USA, October 2000.

[44] Z. Karni, A. Bogomjakov, and C. Gotsman. Efficient compression and rendering of multi-resolution meshes. In *Proceedings of the IEEE Visualization 2002*, pages 347–354, Boston, MA, USA, October-November 2002.

[45] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proceedings of the SIGGRAPH 2000*, pages 279–286, New Orleans, LA, USA, july 2000.

[46] J. Kim, S. Choe, and S. Lee. Multiresolution random accessible mesh compression. *Computer Graphics Forum*, 25(3):323–331, September 2006.

[47] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *Proceedings of the Graphics Interface 2001*, pages 101–110, Ottawa, Ontario, Canada, June 2001.

[48] J. Kim, S. Lee, and L. Kobbelt. View-dependent mesh streaming with minimal latency. *International Journal of Shape Modeling*, 11(1):63–90, June 2005.

[49] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[50] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *Proceedings of the ACM SIGCOMM 2006*, pages 27–38, Pisa, Italy, September 2006.

[51] 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):695–703, 2004.

[52] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *Proceedings of the SIGGRAPH 2000*, pages 131–144, New Orleans, LA, USA, July 2000.

[53] H. Li, M. Li, and B. Prabhakaran. Middleware for streaming 3D progressive meshes over lossy networks. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(4):282–317, 2006.

[54] H. Li, M. Li, and B. Prabhakaran. On supporting high-quality 3D geometry multicasting over IEEE 802.11 wireless networkds. *IEEE Transactions on COMPUTERS*, 58(5):1–14, April 2009.

[55] P. Lindstrom and G. Turk. Image-driven simplification. *ACM Trans. Graph.*, 19(3):204–241, July 2000.

[56] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the SIGGRAPH 1997*, pages 199–208, Los Angeles, CA, USA, August 1997.

[57] N. Magharei and R. Rejaie. PRIME: Peer-to-peer receiver-driven mesh-based streaming. In *Proceedings of IEEE INFOCOM 2007*, pages 1415–1423, Anchorage, Alaska, May 2007.

[58] D. Miyazaki, M. Kamakura, T. Higo, Y. Okamoto, R. Kawakami, T. Shiratori, A. Ikari, S. Ono, Y. Sato, M. Oya, et al. 3D digital archive of the burghers of calais. *Lecture Notes in Computer Science*, 4270:399, 2006.

[59] S. Mondet, W. Cheng, G. Morin, R. Grigoras, F. Boudon, and W. T. Ooi. Streaming of plants in distributed virtual environments. In *Proceedings of the ACM Multimedia 2008*, Vancouver, Canada, October 2008.

[60] S. Mondet, W. Cheng, G. Morin, R. Grigoras, F. Boudon, and W. T. Ooi. Compact and progressive plant models for streaming in networked virtual environments. *ACM Transactions on Multimedia Computing, Communications, and Applications(TOMCCAP)*, 5(3):21:1–22, 2009.

[61] M. Okuda and T. Chen. Joint geometry/texture progressive coding of 3d models. In *Proceedings of the Image Processing 2000*, pages 632–635, Vancouver, BC, Canada, September 2000.

[62] V. N. Padmanabhan, H. J. Wang, and P. A. Chou. Resilient peer-to-peer streaming. In *Proceedings of the ICNP 2003*, page 16, Atlanta, GA, USA, November 2003.

[63] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January 2000.

[64] S.-B. Park, C.-S. Kim, and S.-U. Lee. Error resilient coding of 3D meshes. In *Proceedings of the ICIP 2003*, pages I–773–6, Barcelona, Spain, September 2003.

[65] S.-B. Park, C.-S. Kim, and S.-U. Lee. Error resilient 3-D mesh compression. *IEEE Transactions on Multimedia*, 8(5):885–895, October 2006.

[66] J. Peng, C.-S. Kim, and C.-C. J. Kuo. Technologies for 3D mesh compression : A survey. *ELSEVIER Journal of Visual Communication and Image Representation*, 16(6):688–773, 2005.

[67] J. Peng and C.-C. J. Kuo. Geometry-guided progressive lossless 3D mesh coding with octree (ot) decomposition. *ACM Trans. Graph.*, 24(3):609–616, 2005.

[68] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *Proceedings of the SIGGRAPH 1997*, pages 217–224, Los Angeles, CA, USA, August 1997.

[69] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[70] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the SIGGRAPH 2000*, pages 343–352, New Orleans, LA, USA, july 2000.

[71] S. Rusinkiewicz and M. Levoy. Streaming qsplat: a viewer for networked visualization of large, dense models. In *Proceedings of the SI3D 2001*, pages 63–68, Chaper Hill, NC, USA, March 2001.

[72] R. D. Silva, W. Cheng, D. Liu, W. T. Ooi, and S. Zhao. Towards characterizing user interaction with progressively transmitted 3D meshes (short paper). In *Proceedings of ACM MULTIMEDIA 2009*, Beijing, China, October 2009.

[73] R. Southern, S. Perkins, B. Steyn, A. Muller, P. Marais, and E. Blake. A stateless client for progressive view-dependent transmission. In *Proceedings of the Web3D 2001*, pages 43–50, Paderbon, Germany, February 2001.

[74] G. Taubin. A signal processing approach to fair surface design. In *Proceeding of the SIGGRAPH 1995*, pages 351–358, Los Angeles, CA, USA, August 1995.

[75] G. Taubin. 3D geometry compression and progressive transmission. Technical report, EUROGRAPHICS State of the Art Report, September 1999.

[76] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *Proceeding of the SIGGRAPH 1998*, pages 123–132, Orlando, FL, USA, July 1998.

[77] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Trans. Graph.*, 17(2):84–115, 1998.

[78] D. Tian and G. AlRegib. On-demand transmission of 3D models over lossy networks. *EURASIP Journal on Signal Processing: Image Communication*, 21, June 2006.

[79] D. S. P. To, R. W. H. Lau, and M. Green. A method for progressive and selective transmission of multi-resolution models. In *Proceedings of the VRST 1999*, pages 88–95, London, UK, December 1999.

[80] C. Touma and C. Gotsman. Triangle mesh compression. In *Proceedings of the Graphics Interface 1998*, pages 26–34, Vancouver, BC, Canada, June 1998.

[81] W. Tutte. A census of planar triangulations. *Canadian Journal of Mathematics*, 14:21–38, 1962.

[82] M. Wand. *Point-Based Multi-Resolution Rendering.* PhD thesis, University of Tbingen, Department of Computer Science and Cognitive Science, 2004.

[83] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the IEEE Visualization 1996*, pages 327–334, San Francisco, CA, USA, October–November 1996.

[84] Z. Yan, S. Kumar, and C.-C. J. Kuo. Error-resilient coding of 3-D graphic models via adaptive mesh segmentation. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(7):860–873, July 2001.

[85] S. Yang, C.-S. Kim, and C.-C. J. Kuo. A progressive view-dependent technique for interactive 3-D mesh transmission. *Circuits and Systems for Video Technology, IEEE Transactions on*, 14(11):1249–1264, 2004.

[86] S. Yang, C.-H. Lee, and C.-C. J. Kuo. Optimized mesh and texture multiplexing for progressive textured model transmission. In *Proceedings of the ACM Multimedia 2004*, pages 676–683, New York, USA, October 2004.

[87] Z. Yang, W. Wu, K. Nahrstedt, G. Kurillo, and R. Bajcsy. Viewcast: View dissemination and management for multi-party 3d tele-immersive environments. In *Proceedings of the ACM MULTIMEDIA 2007*, pages 882–891, Augsberg, Germany, September 2007.

[88] W.-P. K. Yiu, X. Jin, and S.-H. G. Chan. Distributed storage to support user interactivity in peer-to-peer video streaming. In *Proceedings of the IEEE ICC 2006*, pages 55–60, Istanbul, Turkey, June 2006.

[89] H. Zhang, O. van Kaick, and R. Dyer. Spectral methods for mesh processing and analysis. In *Proceedings of Eurographics STAR – State of The Art Report*, pages 1–22, 2007.

[90] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. CoolStreaming/DONet: A data-driven overlay network for efficient live media streaming. In *Proceedings of the IEEE INFOCOM 2005*, pages 2102–2111, Miami, FL, USA, March 2005.

[91] Z. Zheng, P. Edmond, and T. Chan. Interactive view-dependent rendering over networks. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):576–589, 2008.

[92] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceeding of the SIGGRAPH 2001*, pages 371–378, Los Angeles, CA, USA, 2001.