University of Wollongong
Research Online

University of Wollongong Thesis Collection

University of Wollongong Thesis Collections

2012

Different approaches to the facilitation of graphics rendering on thin clients based on cloud computing

Dong Nguyen Thanh
University of Wollongong

Recommended Citation

Nguyen Thanh, Dong, Different approaches to the facilitation of graphics rendering on thin clients based on cloud computing, Master of Engineering - Research thesis, School of Electrical, Computer and Telecommunications Engineering, University of Wollongong, 2012. http://ro.uow.edu.au/theses/3783

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

DIFFERENT APPROACHES TO THE FACILITATION OF GRAPHICS RENDERING ON THIN CLIENTS BASED ON CLOUD COMPUTING

A thesis submitted in partial fulfillment of requirements for the award of the degree

Master of Engineering by Research

by

Dong Nguyen Thanh

School of Electrical, Computer and Telecommunications Engineering

UNIVERSITY OF WOLLONGONG

August 2012

This page intentionally left blank

Statement of originality

I, Dong Nguyen Thanh, declare that this thesis, submitted in partial fulfillment of the requirements for the award of Master of Engineering – Research, in the School of Electrical, Computer and Telecommunications Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Dong Nguyen Thanh

August 31, 2012

TABLE OF CONTENTS

List of figures

List of tables

LIST OF ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
FPS	Frame per second
GLX	OpenGL Extension to the X Window System
GPU	Graphics Processing Unit
IBR	Image based rendering
JPEG	Joint Photographic Experts Group
LOD	Level of Detail
MPEG	Moving Picture Experts Group
PC	Personal computer
RFB	Remote Frame Buffer
RTP	Real-time Transport Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VNC	Virtual Network Computing

ABSTRACT

Real-time graphics processing on the cloud poses significant challenges in terms of processing capability, data transmission, and the management of latency. The rendering of large and complex graphics data requires large processing power and significant storage that low-powered machines are unlikely to handle capably. In addition, the transmission of graphics may introduce considerable delays, leading to poor interactivity. Numerous works have been carried out taking these issues into account, most of which being based on level of detail (LOD) and image based rendering (IBR) techniques. However, there are many tradeoffs that need to be carefully studied in order to realize some of the benefits of cloud computing for three dimensional (3D) networked graphics. In this project, we explore the state of the art remote rendering, or in other words, moving the rendering of complex graphics data into a cloud system. A networked rendering paradigm based on our proposed pipeline-splitting method is introduced to facilitate a remote-rendering system with the aim of partitioning the rendering workload between the client and server. We also propose a visibility streaming method for networked applications to reduce the transmission capacity required. One of the main advantages of our proposed methods is that it is easy to scale up at the server side by distributing the workload to be handled in different machines, leading to a significant improvement at the server side in terms of performance.

Acknowledgement

Foremost, I would like to thank my principal supervisor, Professor Farzad Safaei for his motivation, encouragement, immense knowledge, and guidance. This work would not have been possible without his help.

Besides, my special thanks also go to my co-supervisor, Dr Raad Raad for his insightful comments, and assistance.

I am grateful to the staff of the School of Electrical, Computer and Telecommunications Engineering for giving me support and help during my study at the University.

Thanks also to my fellow students and friends, who have helped me during my study at the University.

I would also like to thank my soccer friends for a wonderful time of fun and fitness on every weekend.

Last but not the least, I would like to express my deepest gratitude to my parents and my brother who have supported me during my studies and research projects. Without their encouragement and support, it would have been impossible for me to finish this work.

Chapter 1

Introduction

Chapter contents

1.1. Overview

3D graphics have seen exponential growth since the introduction of the technology through film and popular culture [1-3]. Continued advances in real-time graphics recently led to the birth of numerous real-time multimedia applications on the Internet, namely online games and virtual shopping. There are four primary concerns with respect to graphics applications on cloud computing: (1) Latency: the time it takes to transmit graphics content from the server to the client; (2) Reliability: How often data is lost or corrupted; (3) Bandwidth: How much data can be sent in a given time; (4) End user devices capacity: graphics processing capability as well as battery capacity. All of these considerations, as well as trade-offs, need to be taken into account.

3D graphics rendering places huge processing demands on end-user devices. Different approaches have recently proposed moving this processing load from the user device into the cloud [4-5]. While at first this appears to be a promising proposal, there are many tradeoffs that need to be carefully studied in order to realize some of the benefits of cloud computing for the rendering of 3D graphics. The transfer of graphics content, especially large 3D models [6], over the network may suffer from transmission latency and low bandwidth connections. In addition, graphics processing is considered as a time-consuming process, and in the event that the server has to serve a large number of clients, it may become overloaded; the rendering workload can be shifted into the client to take advantage of its graphics processing capability and relieve the server from the computational burden [7-9]. However, this will further increase the processing demands on the client. In terms of the network, sufficient bandwidth needs to exist to support larger transfers of data between the client and the cloud. The network will also need to be of low enough latency to support real time services and interactivity. From the server side, the servers will also require sufficient processing capacity to handle the 3D rendering in a timely manner.

In this thesis, we study the different approaches to various problems regarding networked graphics applications in cloud computing. Our focus is on solutions that can effectively reduce memory cost, computational workload at client, and network communication overhead. More specifically, we explore the state of the art of remote rendering solutions applied for thin clients which lack the processing power and are memory-limited. A networked rendering paradigm based on our pipeline-splitting method is introduced to facilitate remote rendering system with the aim to split the workload between the client and server. We also propose a method which relies on server-side processing of visibility for 3D mesh streaming. To deal with the substantial workload at the server, we present a parallel framework that can distribute the computational workload at the server side to be processed in different machines. This can lead to a significant improvement at the server side in terms of performance.

1.2. Research motivation

The widespread development of handheld devices in the last decade has increased the demand for multimedia services. Mobile devices are normally not powerful enough to handle computational-intensive applications, thus, the emergence of cloud computing may be important to end-users as it can assist less powerful devices to support computationally expensive applications.

3D graphics are becoming widespread across the Internet, as can be seen in numerous applications such as virtual museums, online games, and other services that use virtual reality or visualization. These applications are carried out not only via personal computers (PCs), but also through various mobile devices. This trend will undoubtedly grow in the years ahead, leading to an increasingly larger amount of data that are to be placed on end-user devices. In this thesis, we aim to find solutions that can reduce the computational workload, and memory cost at the client as well as transmission latency. Therefore, our research has the potential to assist the ubiquitous availability of such applications with the aid of cloud computing.

1.3. Statement of the problem

Interactive 3D applications often require massive computational resources that low-powered devices are unlikely to capably handle. A possible solution is to offload a portion of computations to servers on the cloud, leaving remaining parts to be processed by the client; this can save the client in terms of battery and storage. However, there remains a problem with respect to the transmission of graphics content over the network. Energy efficiency is also a critical problem to mobile devices. Cloud computing can save mobile client energy but in some cases it can lead to more energy usage as the data transmission between the client and the server requires energy.

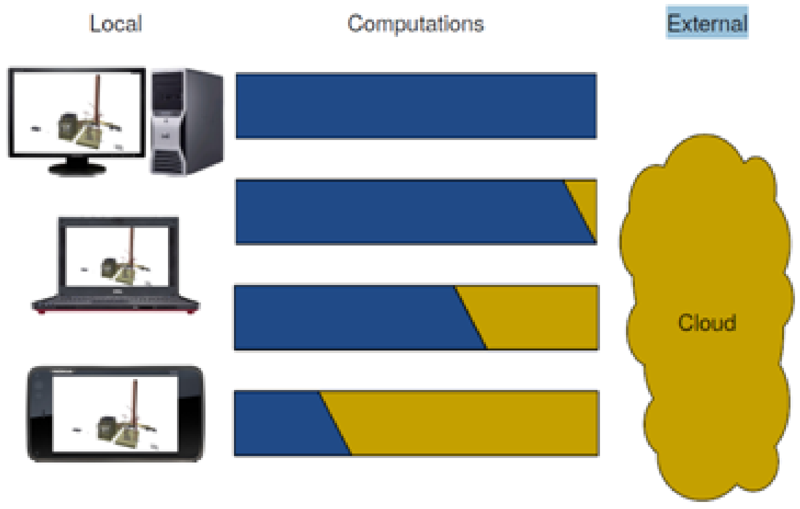


Figure 1. Sharing computations between the client and cloud, the yellow part expressed the computation is performed on the cloud, and the blue part expressed the computation is performed locally [1]

In this thesis, we are going to investigate these tradeoffs in particular cases. Specifically, the following factors are taken into consideration:

1. Transmission latency
2. Rendering capability of the client and the server
3. The transmission of graphics datasets over the network
4. The rendering performance of the system

[1] Project group "Algorithms for 3D rendering using Cloud Computing": http://www.hni.uni-paderborn.de/en/algorithms-and-complexity/teaching/algorithms-for-3d-rendering-using-cloud-computing/

5. Memory cost at the client side
6. Energy consumption at the client

The focus of this thesis is on the methods and techniques that can be applied to deal with aforementioned issues. Therefore, reducing the impact of the transmission latency, memory capacity, and rendering capability of thin clients is considered to be vital in this research.

1.4. Research questions

Regarding networked graphics applications, much work has been done to reduce transmission latency as well as assisting low-powered end-user machines. Most existing approaches focus on reducing graphics data to be processed at the client by simplifying the complexity of the graphics scenes to fit the client's rendering capabilities (level of details techniques) [10]. However, the process of simplification reduces the quality of rendered images and introduces some delay. Therefore, for large-scale scenes, it could be of great benefits to thin clients if all processing, including rendering, is carried out at the server. Image based rendering techniques can be employed to further relieve the client from the rendering workload, making it possible to render very complex 3D scenes on mobile devices. Unfortunately, image quality may suffer due to the limited size of the rendered image and the lack of information to construct new images at novel viewpoints (3D image warping techniques) [11]. In this research, we aim to investigate tradeoffs of existing approaches. Specially, our focus is finding methods which can fulfill the following demands:

1. Methods to split up the workload between the server and the client.

2. Methods to reduce computational workload and memory cost at the client.
3. Methods to reduce transmission latency.
4. The scalability at the server side.

1.5. Thesis structure

The thesis is structured as follows:

- **Chapter 1**: *Introduction.* This chapter presents an overview of our research and its highlighted contributions.
- **Chapter 2**: *Literature review.* This chapter presents background knowledge for our research. Relevant work are also briefly presented and discussed in this chapter.
- **Chapter 3**: *A networked paradigm for remote rendering.* This chapter demonstrates a paradigm for networked rendering pipeline. We also present a remote rendering implementation making use of the paradigm. Our proposed method is then thoroughly compared with other rendering models.
- **Chapter 4**: *A visibility-based streaming framework for networked graphics*. This chapter presents a selective streaming framework that can effectively reduce the underline-processing workload processing at the client as well as reducing the overhead of transmitting data over the network.
- **Chapter 5**: *Conclusion.* The thesis is concluded, and details of future work to be carried out are also presented in this section.

1.6. Thesis contributions

Major contributions of this thesis are listed below:

- A novel method to split up the rendering pipeline is presented aiming to break up the rendering workload from the point that geometry processing is performed at the server, leaving the remaining parts to be done at the client.
- We propose a new networked rendering paradigm for remote rendering based on our pipeline-splitting method. Experimental results show that our method can effectively reduce memory costs and computational workloads at the client.
- We present a study of visibility-based streaming for networked graphics applications. A visibility streaming method also is introduced to support the interactivity between the server and client.
- We also present a parallel framework for visibility streaming that can distribute the computational workload at the server to be processed in different machines. Experiment results demonstrate that our method can reduce memory cost and network communication overhead.

1.7. Publications

The publications arising from this research are listed as listed as follows:

- Dong Nguyen, Farzad Safaei, Raad Raad, "A networked rendering paradigm for remote rendering", Special issue on "Cloud computing", Journal of Software Engineering and Applications, 2012, submitted: 29th August 2012

Chapter 2

Literature review

Chapter contents

2.1. Background knowledge

2.1.1. Mesh representation of graphics

Three dimensional (3D) graphics objects can be presented as a set of polygons or what is so-called polygonal mesh. In this thesis we consider only triangle mesh since it is one of the most prevalent representations of 3D objects. Any none-triangular polygons can be triangulated after a number of simple steps.

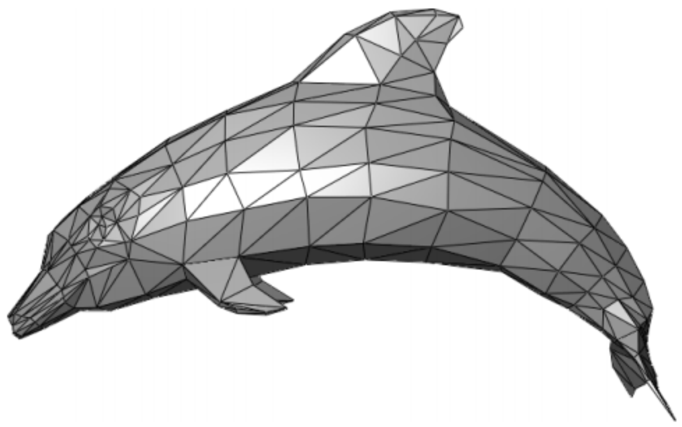


Figure 2. 3D mesh representation of a 3D object[2]

Basically, there are three types of information to present a 3D mesh, including geometry information, connectivity information, and photometry information. We can present a mesh as $M = (V, F)$ where V is a list of vertices $(v_1, v_2, ..., v_n)$ and F is a list of triangles $(tri_1, tri_2, ..., tri_m)$. Each vertex coordinate can be expressed by three floating-point values (x, y, z) and each triangle (or a face) is expressed by three integers referring to three vertices that form the triangle.

2.1.2. Graphics rendering

Graphics rendering is the process of simultaneously generating 2D images from 3D scenes. Graphics data before being displayed on the screen must undergo a number of stages in a so-called graphics rendering pipeline. A graphics pipeline typically consists of a number of stages including vertex processing, primitive assembly, geometry processing, clipping and culling, rasterization, and fragment processing.

[2] [2] Source: http://en.wikipedia.org/wiki/Polygon_mesh

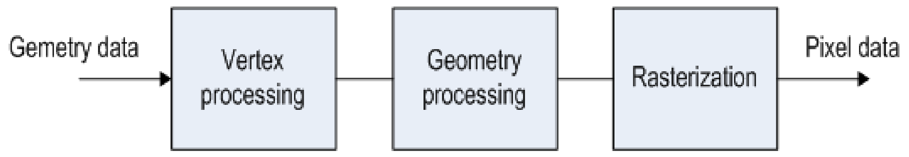


Figure 3. A typical rendering pipeline

Vertex processing: Vertex shaders are responsible for vertex processing by performing operations such as vertex transformation, lighting calculation. The outputs of this stage are individual vertices.

Primitive assembly: In this stage, transformed vertices are grouped based on connectivity information to be converted into primitives (polygons, lines, points).

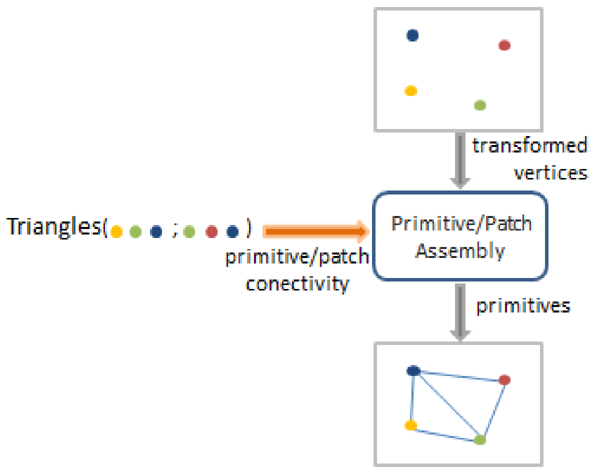


Figure 4. Primitive assembly[3]

Geometry processing: The geometry processing stage happens prior to culling/clipping and after primitive assembly. It receives primitives from previous stage to further process them. Unlike other stages, the geometry stage is capable of generating new primitives from existing ones.

[3] [3] Source: http://www.lighthouse3d.com

Clipping and culling: This stage is responsible for eliminating invisible primitives and those which fall outside of the viewing frustum.

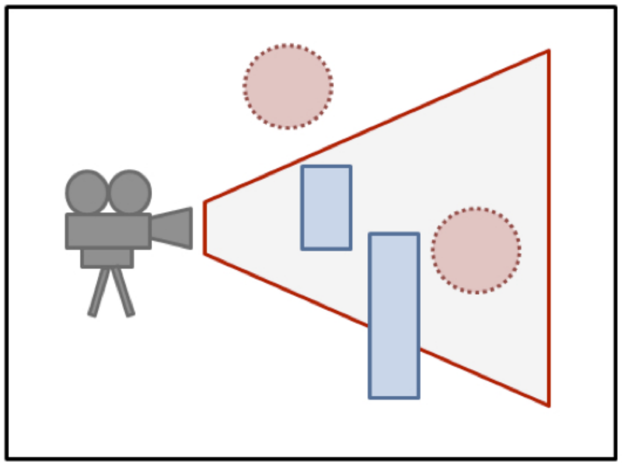


Figure 5. The occluded objects and those which fall outside of the viewing frustum are eliminated[4]

Rasterization: The rasterization stage is responsible for converting every primitive, into a set of fragments. The output of this stage will be passed on to the fragment processing stage for further processing.

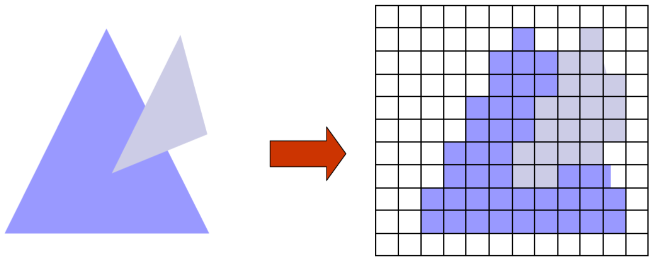


Figure 6. Rasterization stage converting primitives into fragments[5]

Fragment processing stage: A fragment output from the rasterization stage is the size of a pixel, but it is not a real pixel. In fragment processing, data must undergo a

[4] http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1
[5] http://sharavaa.blogspot.com.au/2012/03/graphics-pipeline.html

number of tests (e.g. depth test, stencil test, alpha test), and become a real pixel to be displayed on the screen after getting passed all those tests.

2.1.3. Parallel rendering

Parallel rendering is essential for the rendering of large graphics datasets [12-13] and large tiled display [14]. There has been much work devoted to parallel rendering in the literature. A classification of parallel rendering has been described in [15], in which parallel rendering is classified into sort-first, sort-middle, and sort-last rendering. Recent work on interactive rendering makes use of parallel rendering to achieve a better rendering performance. Lamberti et al. [16] presented a rendering cluster based on Chromium [17-19] to support remote rendering on handheld devices. The system can handle multiple user interactions by making use of a "token" protocol. Parallel rendering can also be applied to volume rendering [20] by dividing volumes into smaller ones and then distributing them to different rendering machines to be handled.

2.1.4. Visibility culling

Visibility culling is extremely essential for the rendering of large and complex 3D scenes. The primary goal of culling techniques is to eliminate primitives that are invisible from the current viewpoint and prevent them from being further processed. This reduces the processing time as this is proportional to the size of remaining visible set. Visibility culling can be roughly classified into view-frustum culling, back-face culling, occlusion culling [21]. Back-face culling [22] culls primitives which face away from the viewer, and view-frustum culling [23] eliminates primitives which fall outside of the viewing frustum, while occlusion culling techniques discard primitives which are occluded by others [24].

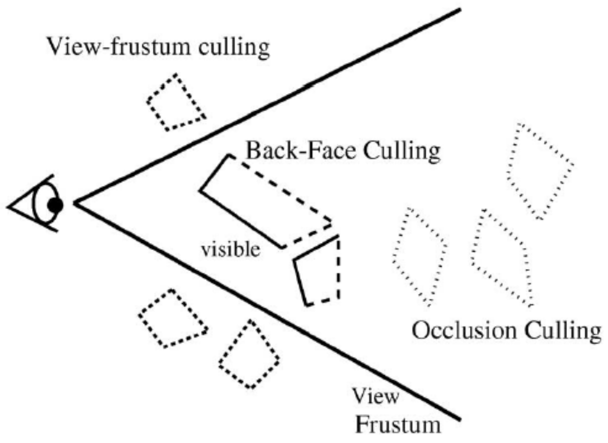


Figure 7. Visibility culling techniques [24]

To date, an enormous amount of work has been done regarding visibility culling techniques. Yoon et al. [25] presented an algorithm for interactive display of complex environments using cluster hierarchies and occlusion culling. Engelhardt and Dachsbache [26] proposed a method for visibility determination of a large number of objects which can improve the rendering performance by culling invisible primitives at geometry shaders.

2.1.5. Computer graphics on thin clients

3D graphics applications on thin devices, especially handheld devices, have been named as one of the fastest growing segments of the graphics industry in recent years. However, there remains a fundamental issue; 3D graphics applications normally necessitate large amounts of computing resources, battery power and storage, while thin devices tend to be limited in these resources. Virtual Network

Computing (VNC[6]) attempts to allocate computing resources to the clients, thus makes it possible to run 3D graphics applications on thin clients. A VNC server stores all rendered images in a frame buffer and sends the content to client on demand through the use of Remote Frame Buffer (RFB) protocol. VirtualGL [27] makes use of VNC protocol for the network streaming of graphics content to thin devices which lack graphics rendering capability. The client sends out OpenGL commands to be processed in the remote Graphics Processing Unit (GPU) at the server side and reads back the rendered images.

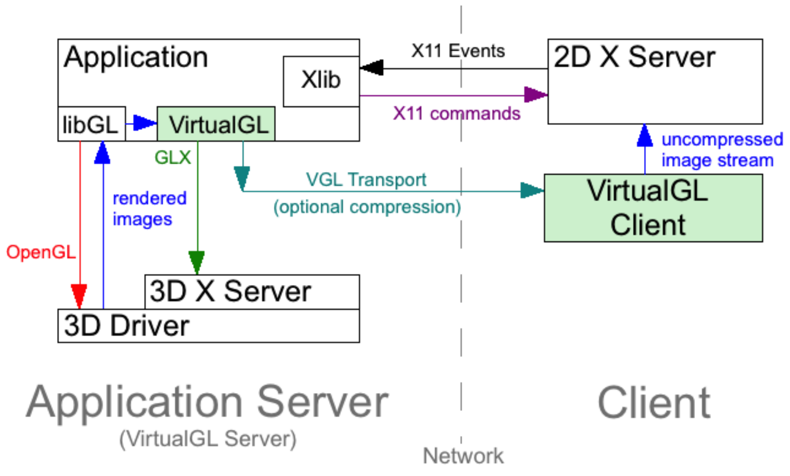


Figure 8. VirtualGL operations[7]

2.1.6. Remote rendering in the cloud

Cloud computing is considered as a promising factor for 3D graphics technologies. The idea is to render graphics data, compress the rendered images at the server side, and send the results to the clients to be further processed. Cloud computing offers great potential in the gaming industry with several solutions in this area. For

[6] http://en.wikipedia.org/wiki/Virtual_Network_Computing
[7] http://virtualgl.org

example, OnLive[8] is known as a cloud gaming solution based on the image-based rendering approach. Graphics data stored in the cloud is rendered and streamed to clients on demand. Similar to OnLive, OTOY[9] provides various types of real-time graphics services in the cloud such as computer applications, video games, High Definition media content, and film/video special effect graphics through server side rendering.

There has been a great deal of attention paid by researchers to remote graphics rendering using cloud computing infrastructure. Okamoto et al. [28] introduce an interactive rendering system for large 3D mesh models based on cloud computing. The system makes use of both image-based rendering and model-based rendering techniques to balance the workload between the client and server. Winter et al. [29] propose a hybrid approach to facilitate graphics processing on thin clients. An adaptive mechanism is proposed to select an appropriate transmission method according to the scenarios of scenes. Jurgelionis et al. [30] introduced a hybrid approach based on Game@Large [31-32]. This solution is fairly flexible since it can support both low- and high-powered devices concurrently by applying two streaming approaches. For small displays like handheld devices, the server performs the rendering tasks and streams rendered images and audio data to the client. For high-end devices, the client possesses its own graphics processing unit and is capable of performing rendering by itself; hence graphics commands are encapsulated and transmitted to the client to be processed locally.

[8] http://www.onlive.com
[9] http://www.otoy.com/

2.1.8. Energy efficiency for mobile devices

The need for energy efficiency is very critical for mobile devices since the advance of battery technology is insufficient to meet the demand of mobile users. Cloud computing has tremendous potential to save mobile energy. However, the tradeoff between energy consumed by computation and the energy consumed by communication needs to be carefully considered. Miettinen and Nurminen [33] pointed out that there must be a break-even point for computation offloading. For the sake of efficiency, the energy consumed by the local computation (E_{local}) must not exceed the energy consumed by communication (E_{cloud}), or in other words $E_{cloud} < E_{local}$. Let D be the amount of Data to be transferred in bytes and C be the computation for the workload in CPU cycles, we have: $E_{cloud} = \frac{D}{D_{eff}}$ and $E_{local} = \frac{C}{C_{eff}}$. Where D_{eff} and C_{eff} are device specific data transfer and computing efficiencies. To be beneficial, the following inequality must hold:

$$\frac{C}{D} > \frac{C_{eff}}{D_{eff}}$$

An analysis presented by Karthik and Yung-Hsiang Lu [34] indicates that the energy saved by computation offloading depends on wireless bandwidth (B), the amount of computation to be performed (C), and the amount of data to be transmitted (D). According to the analysis, the energy saved can be calculated as follows:

$$\frac{C}{M}\times(P_c - \frac{P_i}{F}) - P_{tr}\times\frac{D}{B}$$

Where: C is the number of instructions required for the computation, M is the speed, in instructions per second, of the mobile. The speed of the server is F time faster than that of mobile device (S = F x M). P_c, P_i, P_{tr} respectively are the energy consumption, in watts, for computing, while idle, and for sending and receiving data. This indicates that not all applications are energy efficient when migrated to the cloud, at some point it is more efficient to perform the computation locally rather than remotely.

2.2. Client-server rendering architecture

Graphics processing in client/server architecture can be roughly divided into three categories: client-side method, server-side method, and hybrid method [35-37].

2.2.1. Client-side method

In this method, the server simply sends graphics data to the client and the client is responsible for rendering the entire 3D models. The conventional method of client-side rendering involves transmitting graphics commands to the client and is to be processed locally [38-39]. This method can reduce workload at the server, but it increases the processing demand on the client. This is suited for small applications, but is inappropriate for complex applications that require high rendering power. Moreover, graphics data to be transmitted to the client may be large leading to a long downloading time. To make it possible for the transfer of larger models, the server may perform the simplification and conversion operations to calculate a progressive representation composed of a simplified model and a series of mesh refinements that the client will progressively download and display [40-42].

2.2.2. Server-side method

In contrast to the client-side method, this method involves the server as completely responsible for graphics processing. The server renders the 3D scenes and transmits the rendered images to the client to be displayed [43-45]. This is highly beneficial to thin clients which often lack specialized hardware and are memory-limited [16, 46-47], such as mobile devices [48-49]. However, the limitation of this method is that the server may become congested when serving a large number of clients and an appropriate network connection, that is, sufficient bandwidth, need to exist. This may be fine for fixed type networks, but may not be appropriate for wireless networks. In addition, the latency due to the constant transmission of rendered images from the server to client may reduce responsiveness and interactivity. Image based rendering (IBR) techniques can be implemented in the client to improve frame rates and to deal with the transmission delay [11, 50]. However, there are some tradeoffs between the image quality and transmission latency [45].

2.2.3. Hybrid method

In this method, both the client and server get involved in the rendering process. Rendering tasks are partially accomplished at the server and the remainder is performed at the client. Therefore, the rendering workload can be shared between the server and client [37, 51]. However, deciding which parts to be performed at the client and which parts to be performed at the server is not an easy task. Noguera et al. [37] proposed a technique to split the rendering workload between the server and the client based on the view volume. The client is responsible for rendering the terrain which is close to the viewer and the server renders the terrain far away from the viewer. Diepstraten et al. [52], in a different manner, split the process of image generation in order to balance workload between the client and the server. The server partially renders the 3D scene and sends 2D primitives to be processed on the client.

However, this may lead to the downgrading of image quality since the client has to rely on feature lines to draw the image.

2.3. Graphics streaming

The transmission of graphics datasets through the network is considered a major bottleneck due to the bandwidth limitation and the size of data to be sent. In this section, we consider two ways of transmitting graphics data from the server to the client: Image-based streaming, and Mesh streaming.

2.3.1. Image-based streaming

Image-based streaming has been widely used in remote rendering system [53-54]. Panka et al. [55] proposed a framework to facilitate remote visualization on mobile devices, in which graphics data is rendered at a the server side, the rendered images then are compressed and streamed to the client as a video stream. Boukerch et al. [46-47] have presented a rendering method based on image-based rendering technique to assist the streaming of images over the network. A packetization scheme and a feedback mechanism have also been proposed to deal with the variations of the wireless network bandwidth.

Compression is highly essential for image-based streaming to make effective use of network capacity for the streaming of complex 3D scenes over the network [56]. Various compression techniques for graphics streaming have been considered in the literature. For example, Cortelazzo and Zanuttigh [57] present a predictive compression scheme making use of JPEG and JPEG-2000 for remote visualization based on image-based rendering techniques. Constantinescu and Vlădoiu [58] proposed an adaptive compression method for remote rendering, an appropriate compression scheme according to the variation of frame rate is selected from different ones, for example, ZLIB, LZO, BZIP2, RLE, to be used.

MPEG-4 [59-61] is used for image-based streaming systems, this trend becomes promising especially as most handheld devices are capable of decoding MPEG-4 [62]. Liang Cheng, et tal., [63] make use of MPEG-4 streaming for their remote rendering system. They also propose a fast motion estimation algorithm to assist the encoding process.

2.3.2. Mesh streaming

In contrast to image-based streaming, in mesh streaming, the geometric data is streamed to the client to be rendered. There has been a growing body of research with respect to streaming mesh over the network. Progressive mesh (PM) streaming [64] aims to minimize the transmission cost and rendering cost at the client. In this method, a coarse model is first sent to the client, and then a series of refinements will be streamed to improve the image quality [65]. Therefore, this reduces the waiting time as it can enable interactions without a full download of data.

To construct a PM representation, the original mesh $\hat{M}$ is simplified through a series of edge collapses (*ecol*) to yield a much simpler base mesh M^0 and a sequence of refinements. The simplified model (M^0) is first transmitted to the client, and progressive meshes then refine the object by the continual transmission of refinements.

$$\hat{M} = M^n \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0$$

At the client side, an inverted sequence, the so-called vertex split (*vsplit*), is employed to refine the model from the coarse model to original one.

$$M^0 \xrightarrow{\text{vsplit}_0} M^1 \xrightarrow{\text{vsplit}_1} \ldots \xrightarrow{\text{vsplit}_{n-1}} M^n = \hat{M}$$

Progressive mesh streaming can achieve good interactivity however it may suffer from low-quality of images.

In view-dependent streaming, the server progressively streams geometry data to the client with respect to the current viewing parameters. Rusinkiewicz and Levoy proposed a view-dependent streaming method based on QSplat[10] [66] to facilitate the streaming of complex 3D models. Yang et al. [67] introduced a patch-based view dependent streaming technique. First, the mesh is partitioned into a number of patches which are compressed offline and streamed to the client on demand. The client relies on received patches and the connectivity information to perform the rendering by itself. However, one drawback of the method is that it causes an unsmooth change at the client side due to the alternation of patches. Schneider and Martin [36, 68] have proposed an adaptive framework for the transmission of graphics data in the client/server environment. A number of factors are taken into account, such as network conditions, user preferences and the rendering capabilities of the client and server in order to select an appropriate transmission method to stream 3D models to the client.

[10] http://graphics.stanford.edu/software/qsplat/

Chapter 3

The networked paradigm for remote rendering

Chapter contents

3.1. Introduction

In recent years, networked three dimensional (3D) applications have become more demanding in terms of processing capacity. Geometry processing including vertex transformations, lighting calculations and triangle assembly appears challenging due to the complexity of 3D models and restricted capabilities of graphics hardware in mobile devices – otherwise known as a thin client. Therefore, it is expected to take advantage of cloud computing for the computation of a portion of the rendering tasks, leaving remaining tasks to be computed by the client.

To date, existing approaches to graphics rendering on thin clients make use of various techniques such as mesh compression [69-70], mesh simplification [71] to assist the rendering of huge mesh on mobile devices. However, there remain some major disadvantages of using such techniques. First, the mobile device is required to be capable of performing the rendering by itself. Secondly, there must be a trade-off between the frame rate and image quality [72-73]. Although image-based rendering techniques can be of great use to facilitate the rendering on mobile devices at a relatively low cost, it appears inappropriate to such systems that require a full control of image size. Additionally, there remains an issue regarding the image quality due to the use of 3D warping techniques [11, 50].

In this chapter, we provide an approach to graphics rendering on thin clients. Our approach attempts to reduce the computational workload and memory cost at the client. We develop a hybrid framework, in which both the server and the client get involved in the rendering process. First, a pipeline-splitting method is proposed with the aim of decoupling the geometry processing stage from the rendering pipeline. Different from conventional pipeline-splitting methods, our approach relies on transform feedback mode[11] to obtain data from the buffer object in the graphics card. This achieves hardware acceleration for geometry processing while hardware support still remains available for the rasterization stage as soon as the data is put back to be rasterized in the graphics card. Next, we introduce a networked paradigm for remote rendering based on our pipeline-splitting method. A theoretical analysis is presented, and then an implementation based on client/server architecture is built to investigate the proposed paradigm. The experimental results shown that our method can reduce memory cost and computational workload at the client and the processing time at the

[11] http://www.opengl.org/registry/specs/NV/transform_feedback.txt

server. Moreover, as the rasterization stage is executed at the client, our approach gives the end users full control of the image size on the screen.

3.2. Rendering pipeline analysis

In general, a rendering pipeline typically consists of a number of stages including vertex processing, geometry processing, rasterization, and fragment processing. For the sake of simplicity, we consider the pipeline with only two separated stages. The first stage named geometry processing is responsible for vertex transformations, lighting calculations, and triangle assembly. The second stage named rasterization is a combination of clipping/culling, rasterization, and fragment processing.

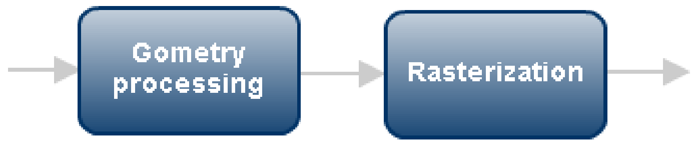
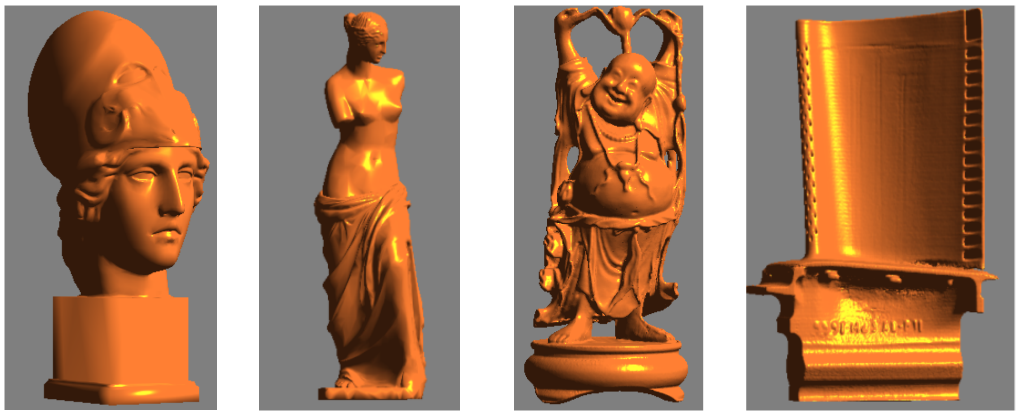


Figure 9. A typical rendering pipeline

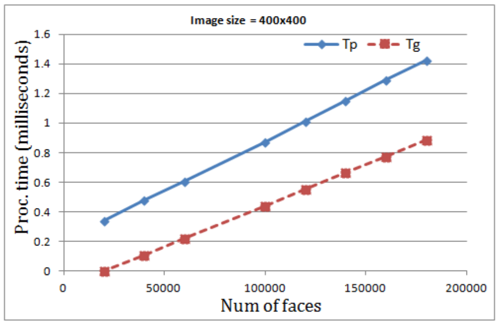
From this perspective, we will present an analysis of the rendering pipeline in terms of processing time. It is worth noting that the determination of the most time-consuming stage in the graphics rendering pipeline is challenging as each stage depends on various factors. For example, the processing time at the geometry processing stage depends on the number of primitives while the processing time at rasterization stage depends on the number of input primitives, the viewing angle, and the image resolution.

For clarity, let T_p be the processing time of the entire pipeline, and T_g be the processing time of the geometry processing stage. The total execution time T_p is equal to the sum of the execution times for the two stages: geometry processing and rasterization. T_g can be roughly estimated by disabling rasterization stage to prevent primitives from being rasterized. Note that we do not take into account the time taken to clear and swap the buffer during the rendering for the sake of simplicity.

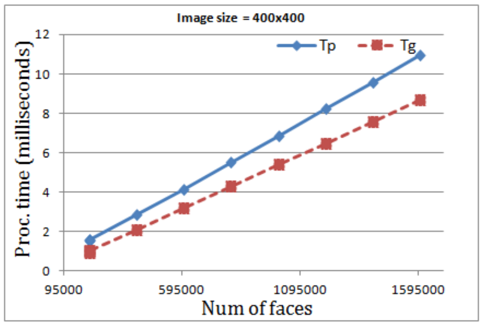


(a) (b) (c) (d)

Figure 10. A set of 3D models are used in the test, (a) Atenean - 7546 vertices, 15014 triangles, (b) Venus - 19847 vertices, 43357 triangles, (c) Happy - 399864 vertices, 800000 triangles, (d) Blade - 800124 vertices, 1599996 triangles

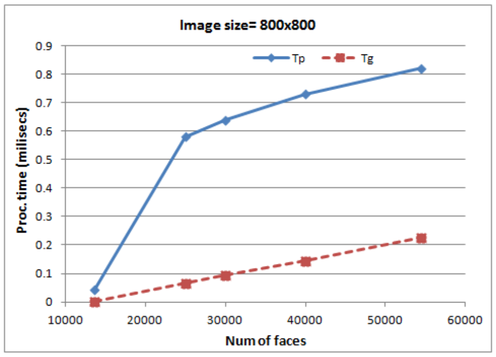


(a)

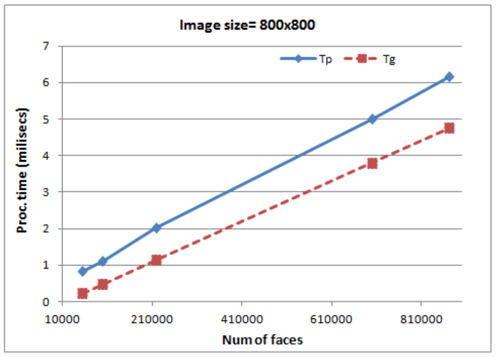


(b)

Figure 11. Processing time at geometry processing stage compared to the rendering time – the test was done on NVIDIA Geforce 9500 GT – (a) tested with 3D models with number of faces is less than 200k, (b) tested with 3D models with number of faces is less than 1600k

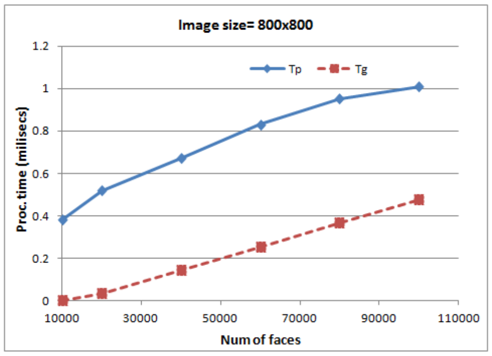


(a)

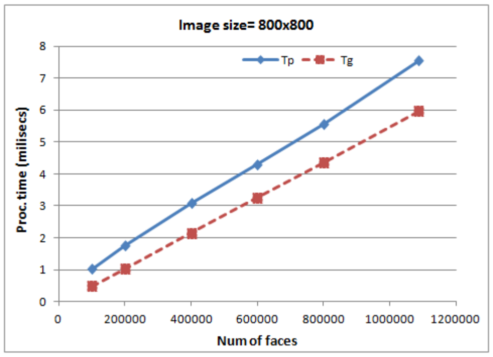


(b)

Figure 12. Processing time at geometry processing stage compared to the rendering time in case of dragon model – graphics card: NVIDIA Geforce 9500GT (a) the number of faces is less than 100K (b) the number of faces is less than 1M



(a)



(b)

Figure 13. Processing time at geometry processing stage compared to the rendering time in case of happy model – graphics card: NVIDIA Geforce 9500GT (a) the number of faces is less than 100k (b) the number of faces is less than 1200k

We investigate the impact of the image resolution and the number of primitives to the processing time at geometry processing stage and the rendering time of the entire pipeline. Figure 12, 13, 14 demonstrate some experimental results obtained from the test. It further indicates that for complex 3D models and small image size, tremendous amount of time is spent at geometry processing stage. Therefore, it is desirable to offload the geometry processing stage to a dedicated server, and the rasterization stage is handled at the client. This can balance the rendering workload between the client and the server to some extent.

3.3. Networked rendering framework

In this section, we describe a scheme for remote rendering based on our pipeline-splitting method. At first, we present a paradigm for a networked rendering pipeline that extends the traditional rendering pipeline to include network transmission of geometry data. The rendering pipeline is divided so that some stages of it are offloaded to the remote server and the remainders remain at the client.

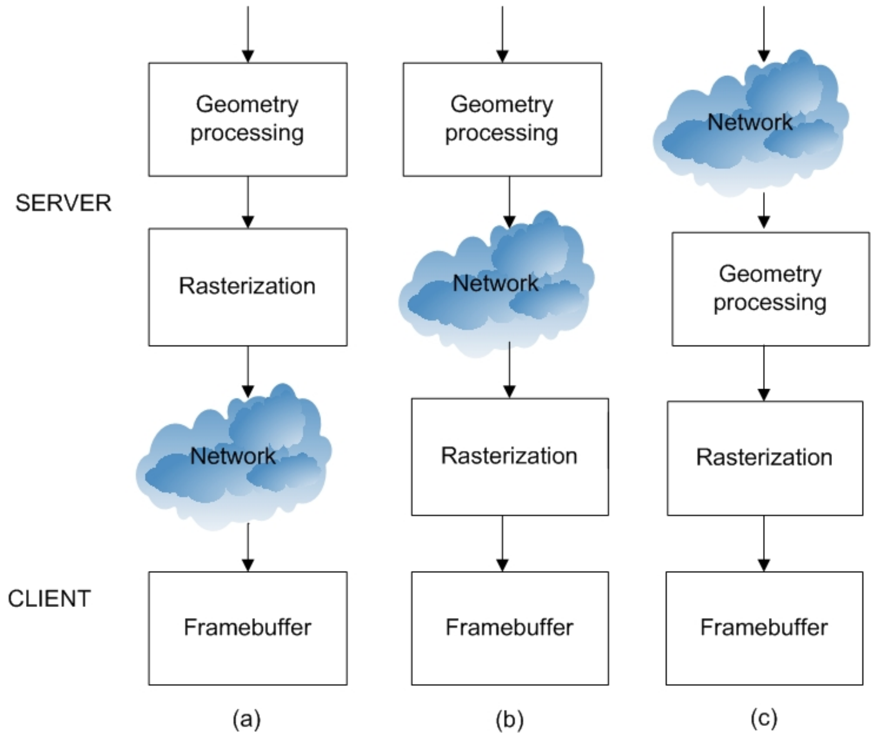


Figure 14. Different architectures of networked rendering pipeline, (a) the entire pipeline is placed on server, (b) geometry is placed on server, rasterization is on client, (c) the entire pipeline is placed on client

3.3.1. Pipeline splitting method

Typically, the rendering pipeline resides on a single machine. It is difficult to divide a graphics rendering pipeline into stages due to the tight coupling of the geometry and rasterization stages. The idea of breaking rendering pipeline has existed for some time. Williams et al. [74-75] proposed a method to separate the geometry stage and rasterization stage by adding two extensions to OpenGL library: triangle-feedback and triangle-rasterize. The triangle-feedback function passes all primitives through the geometric portion without rasterizing them while the triangle-rasterize function takes the data from geometric portion and put it into rasterization stage. To achieve hardware acceleration for rasterization, a vertex program is implemented to pass primitives into the hardware rasterizer on the graphics card. Graphics hardware acceleration, however, remains undone for geometry processing. Banerjee et al. [76-77] combined Mesa3D[12] and socket networking code together to build RMesa (Remote Mesa) which can break the rendering pipeline into sub stages. The client can offload some stages in the pipeline to the remote server to be processed and then get the result back. Unfortunately, the approach offers no graphics hardware-acceleration for both geometry processing and rasterization.

We take a different approach to split the rendering pipeline based on transform feedback mode. The use of transform feedback makes it possible to capture vertex attributes of the primitives processed by geometry processing stage. Vertex attributes are selected to store in a buffer, or several buffers separately which can be retrieved some time later. The rest of pipeline can be discarded by disabling rasterization stage to prevent primitives from being rasterized. This way uncouples geometry processing

[12] http://www.mesa3d.org/

stage from rasterization stage. The transformed primitives copied from transform feedback buffer then can be rasterized in a different machine. Note that the entire process happens inside the pipeline, therefore our method supports hardware-acceleration to both geometry processing and rasterization stage.

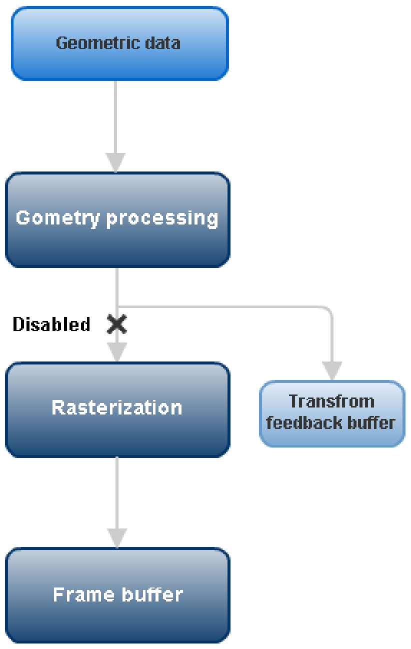


Figure 15. Transform feedback operation – vertices are transformed and stored in the transform feedback buffer object which can be obtained in the middle

3.3.2. Remote rendering based on the pipeline-splitting method

We now introduce a remote rendering framework making use of the pipeline-splitting method that we have presented earlier. The basic concept is similar to image-based rendering, the major difference is that the sever sends back transformed primitives instead of rendered images to the client.

Table 1. Notation 1

Symbols	Quantity
F	List of faces constructed the mesh
F_c	The remaining faces after culling
M, N	The number of faces stored in F and F_c respectively
CHUNK	Number of faces stored in a packet
p	Number of packets to be sent to the client

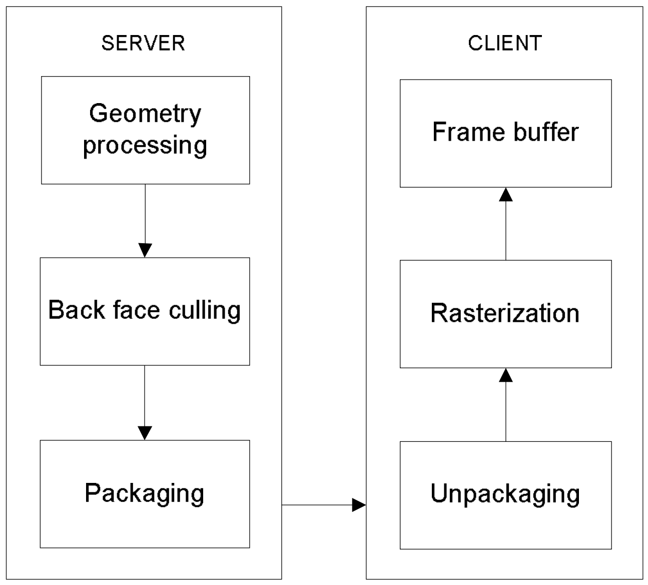


Figure 16. Client-server architecture for the proposed framework

In our proposed framework, the server performs geometry processing on demand according to the viewing parameters received from the client. The back-face culling method [22, 78] then is employed to cull invisible primitives from transformed ones. The remaining primitives then are packaged to be sent to the client for rasterization.

To deal with restrictions in network performance and bandwidth, we take into account the network protocol for the data transmission. For the sake of transmission efficiency, it is important that UDP is employed for data transmission and TCP is used for exchanging messages and commands. To further reduce the latency, graphics content is packetized or can be compressed prior to the transmission. A chunk of primitives is grouped in a packet to be sent to the client for further processing. The number of packets to be sent for the rendering of a frame can be calculated as follows:

$$p = \lceil M/CHUNK \rceil = \lceil \alpha N/CHUNK \rceil \qquad \text{(Equation 1)}$$

Where $\alpha = M/N$ is culling ratio ($0 < \alpha \leq 1$). It is worth noting that the value of α depends on the shape of the 3D model and the position of the model corresponding with the camera.

Table 2. The average value of α tested with several 3D models

Model	**The average value of α**
Shark	0.445504
Beethoven	0.575944
Car	0.500286
Ateneum	0.526975
Dragon	0.429286
Bunny	0.498222

a. Transmission latency

Supposed that the time taken to transmit a packet to the client is t_p. t_p depends on network capacity (bw) and the size of packet (s_p): $t_p = s_p / bw$.

Table 3. Time to transmit a packet

CHUNK	**t_p (secs)**	
	10 Mbps	**100 Mbps**
600	0.03456	0.003456
300	0.01728	0.0017728
200	0.01152	0.001152
100	0.00576	0.000576

Let T be the transmission time of all primitives after performing back-face culling. This is equivalent to the transmission of p packets:

$$T = p \times t_p = \lceil \alpha N / CHUNK \rceil \times (s_p / bw) \qquad \text{(Equation 2)}$$

It can be seen that the transmission latency is linearly proportional to the number of faces N.

Table 4. A theoretical estimation of the time it takes to transmit 3D models with different level of details (α = 0.5)

N	p (CHUNK = 600)	T (secs)	
		10 Mbps	100 Mbps
10000	8	0.27648	0.027648
20000	17	0.58752	0.058752
40000	34	1.17504	0.117504
60000	50	1.728	0.1728
80000	67	2.31552	0.231552
100000	84	2.90304	0.290304

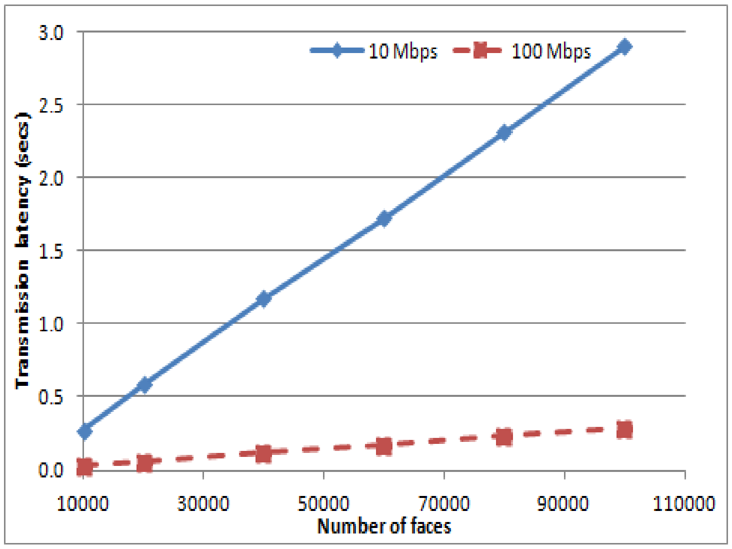


Figure 17. A theoretical analysis of transmission latency (α = 0.5)

b. Bandwidth requirements

Table 5. Notation 2

Symbols	**Quantity**
τ	Time to send a request to server
t_s	Processing time at the server
T	Time to transmit data to the client
t_c	Processing time at the client
$\sum t$	The total amount of time for a frame

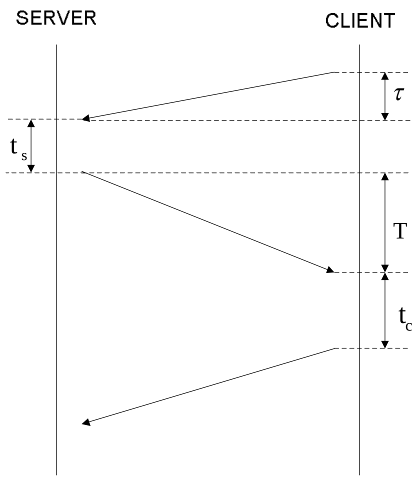


Figure 18. Analytical cost model of the proposed framework

The main limitation of our framework, however, is the network connection between the server and the client. With the help of culling process, the amount of data has been reduced significantly. However, it might take a considerable amount of time to transfer data over the low-bandwidth network causing poor interactivity. Therefore, there must be a trade-off between the frame rate and the network capacity. The question then is how much bandwidth is needed to achieve a frame rate of FPS. This results in an essential upper-bound on the total processing time which should not be greater than 1/FPS.

$$\sum t = \tau + t_s + t_c + T \leq 1/FPS \qquad \text{(Equation 3)}$$

The server is assumed to be very powerful and the size of request data is very small so that τ and t_s is very small, therefore:

$$\sum t \approx t_c + T \leq 1/\mathrm{FPS} \qquad \text{(Equation 4)}$$

Substituting the earlier obtained equations we have:

$$\lceil \alpha N/\mathrm{CHUNK} \rceil \times (s_p/bw) + t_c \leq 1/\mathrm{FPS} \qquad \text{(Equation 5)}$$

Thus:

$$bw \geq (s_p \alpha N/\mathrm{CHUNK})/(1/\mathrm{FPS} - t_c) \qquad \text{(Equation 6)}$$

Denote $bw_0 = (s_p \alpha N/\mathrm{CHUNK})/(1/\mathrm{FPS} - t_c)$, we can see that bw_0 depends on the total number of faces and the rendering capability of the client.

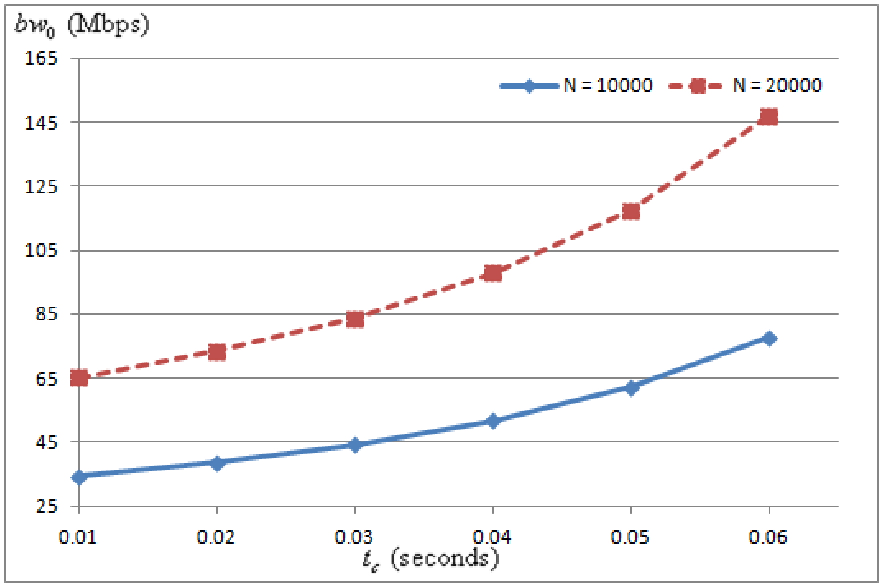


Figure 19. Bandwidth requirements in case $\alpha = 0.5$, $\mathrm{CHUNK} = 600$, $\mathrm{FPS} = 10$, t_c **varies**

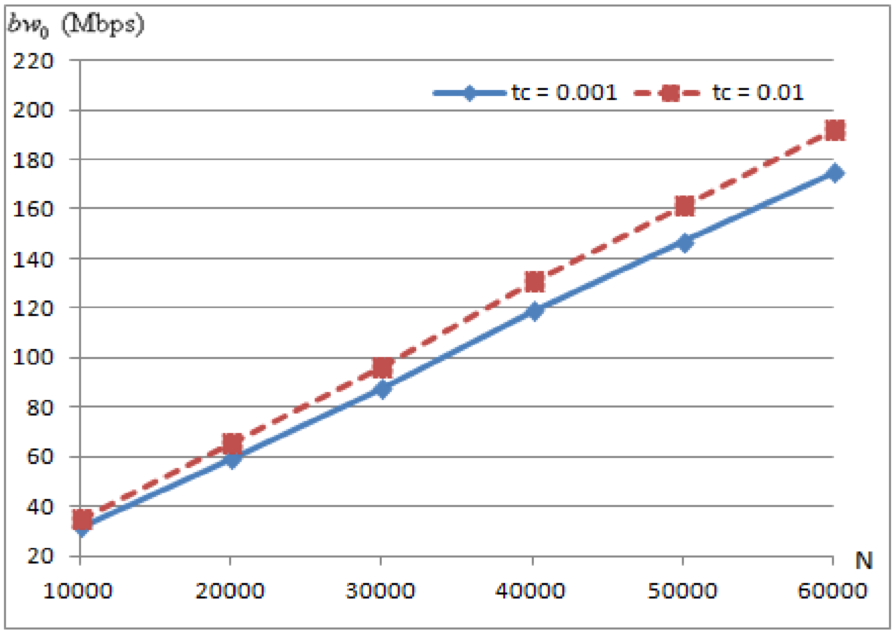


Figure 20. Bandwidth requirements in case $\alpha=0.5$, $\text{CHUNK} = 600$, $\text{FPS} = 10$, N **varies**

3.3.3. Parallel geometry processing

In terms of the performance at the server side, it is expected to perform geometry processing in parallel. The advantages of parallel processing are twofold. On the one hand, it speeds up the processing at the server side. On the other hand, it enhances the system capacity to be capable of serving multiple clients concurrently. In this section, we present a framework for parallel geometry processing. We extend our networked rendering paradigm by dividing the total number of primitives per frame by the number of available server.

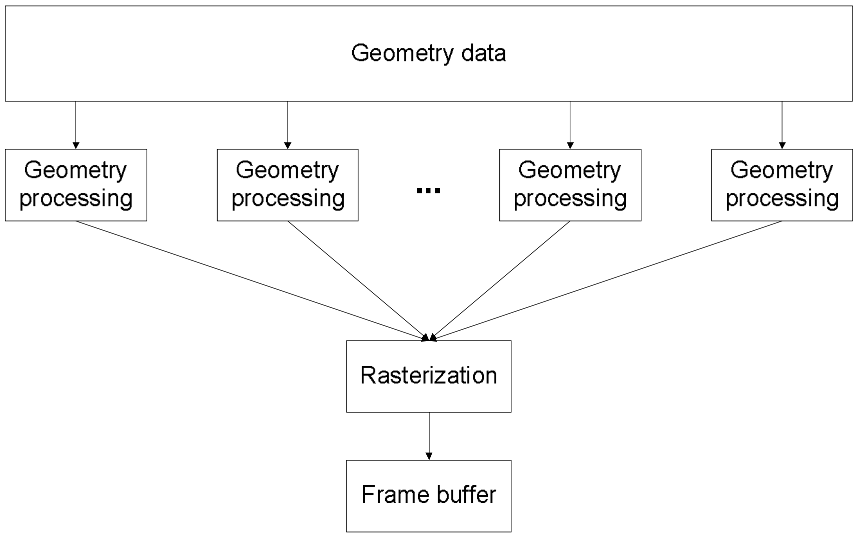


Figure 21. Parallel geometry processing framework

Table 6. Notation 3

Symbols	Quantity
F	The original mesh which consists of N primitives $F = \{f_1, f_2, ..., f_N\}$
$F_1, F_2, ..., F_M$	Sets of primitives decoupled from F
$F_{c_1}, F_{c_2}, ..., F_{c_M}$	Sets of remaining primitives after culling

The operation can be briefly described as follows. The 3D mesh F is first divided into sets of primitives: $F_1, F_2, ..., F_M$ which are to be handled in M servers respectively ($F = \bigcup_{i=1}^{M} F_i$, and $F_i \cap F_j = \varnothing \; \forall 1 < i < j < M$). Each server in the parallel framework operates similarly to a single server in the networked paradigm that we have presented earlier. Consequently, the outputs of the servers are sets of visibly transformed primitives $F_{c_1}, F_{c_2}, ..., F_{c_M}$ which are then transmitted to another machine for rasterization on the same basis. The client in turn receives F_{c_i} from the servers and performs the rasterization stage as soon as all data has been received. The method, of course, can speed up the geometry processing as the geometry processing computation is processed in parallel in different machines. However, it is worth noting that as the client side is not scaled up, the processing time at the client remains unchanged.

The following presents some images obtained from our tests with the distributed geometry processing framework. First, we use two servers for geometry processing. The client receives transformed primitives from both servers and performs rasterization by itself. The second test with three servers operates on the same basis, except the workload now is to be handled in three different servers.

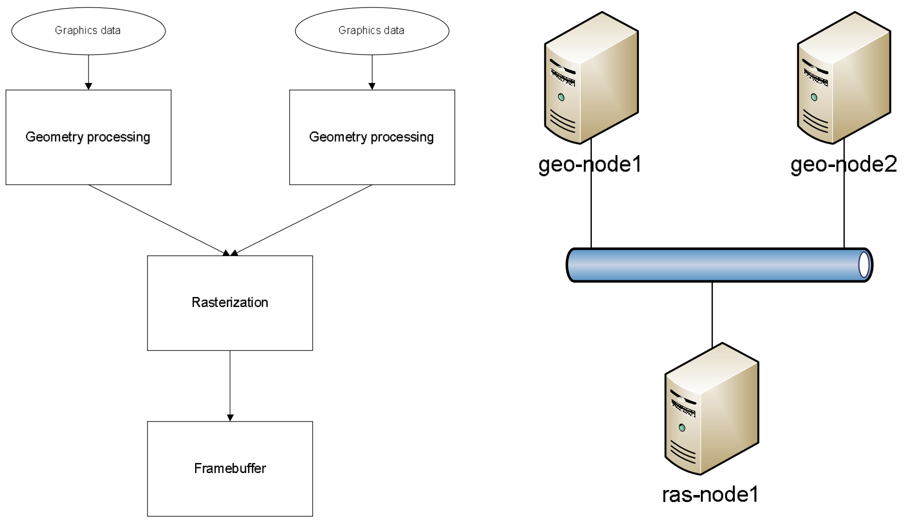


Figure 22. Parallel framework with two servers

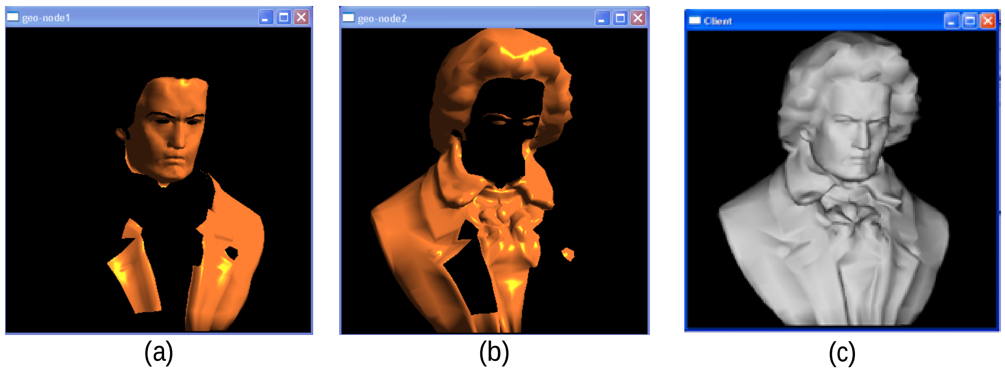


Figure 23. (a) Geometric data processed at geo-node1 (no rasterization discarded), (b) Geometric data processed at geo-node2 (no rasterization discarded), (c) Rasterization is done at the client

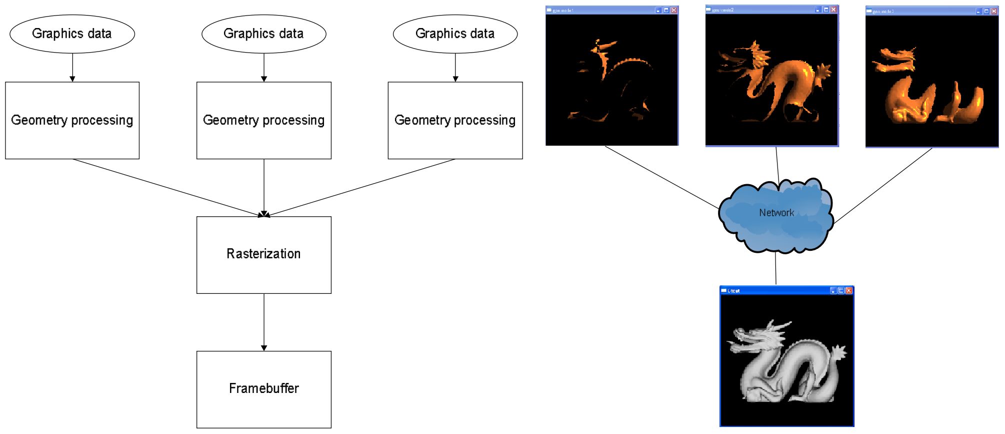


Figure 24. Geometric data is distributed to 3 servers for geometry processing, the transformed primitives then is transmitted to the client to be rasterized there

This parallel framework can be applied to sort-middle parallel rendering as the rasterization stage is parallelized to be performed in different machines. In sort-middle parallel rendering, geometry processing and rasterization are performed on separate processors in many systems, which has been found to be the most natural place to break up the pipeline.

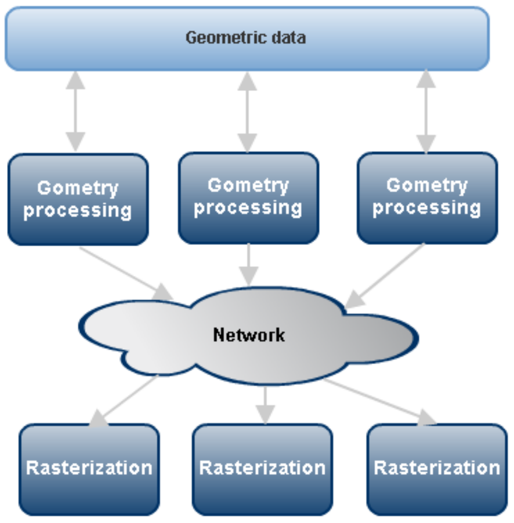


Figure 25. Our parallel framework can be applied to sort-middle rendering

3.4. Experimentation

We have implemented a remote rendering system on Windows in C++ using OpenGL making use of the proposed pipeline-splitting method to split the rendering workload between the server and client. The server we used in the test is Intel ® Core ™ i7 CPU, 3.24 GB of RAM, with NVIDIA GeForce 9500. A DELL T6600, Intel® Core™ 2 Duo CPU 2.2 GHz, 2G RAM is used as a client.

a. Processing time in the pipeline

We make a comparison between local rendering and our method in terms of processing time in the rendering pipeline at the client side. It shows that, our method can reduce the processing time at the client significantly, especially for 3D models with high levels of detail, as the number of faces processing at client has been reduced and the geometry processing stage has been performed at the remote server.

Table 7. A comparison between our proposed method and local rendering in terms of processing time

Model	Num of verts	Num of faces	Local rendering (milliseconds)	Our method (milliseconds)
Beethoven	2521	5030	4.2	2.7
Car	5247	10474	7.2	4.8
Ateneam	7546	15014	10	6.0
Dragon	10006	20000	17	8.0
Venus	19847	43357	32	18
Bunny	34834	69451	48.6	27.6

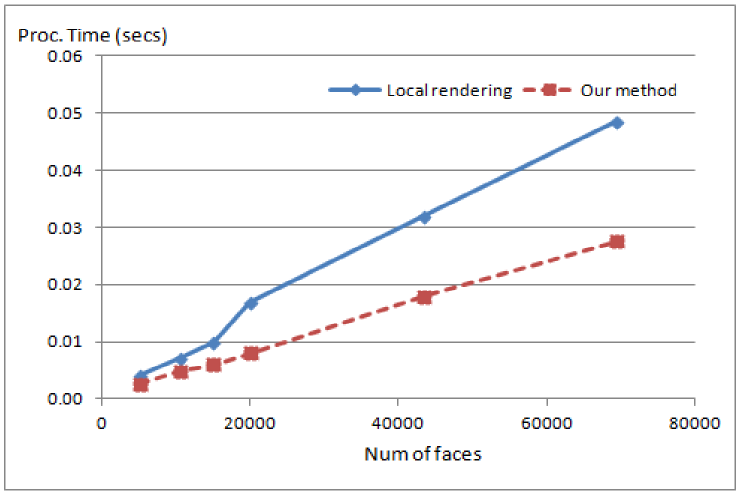


Figure 26. Comparison between our method and local rendering in terms of processing time at client (image size = 400x400)

We also compare our method with server-side rendering in terms of processing time at the server. We take into account the time taken to copy data out of the pipeline. For example, in the case of server-side rendering, we measure the processing time of the entire pipeline plus the time taken to copy data from the frame buffer to CPU. And in our method, we measure the processing time at geometry processing stage and the time to copy data from the transform feedback buffer. When the number of primitives to be processed is small and the image size is large, the processing time at the server is significantly reduced in our method compared to that of server-side rendering. Note that when the fragment processing is relatively cheap, the transform feedback could end up being a major bottleneck leading to more processing time at the server in our method compared to that of server-side rendering.

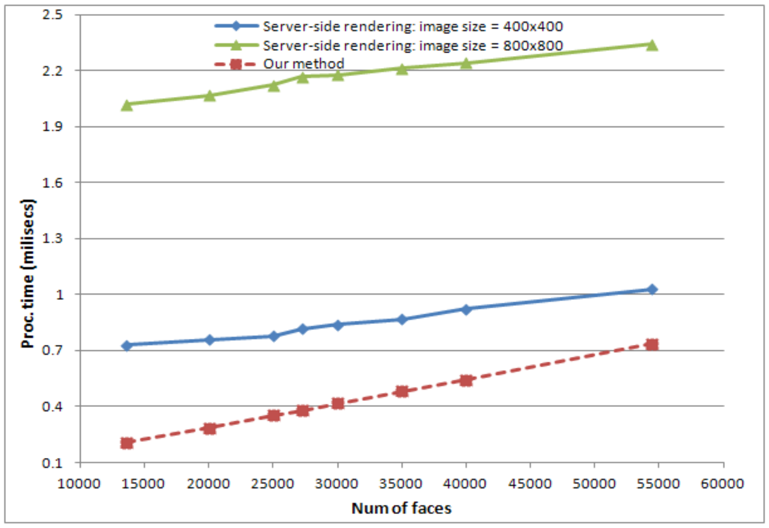


Figure 27. A comparison between server-side rendering and our method in terms of processing time tested with dragon model

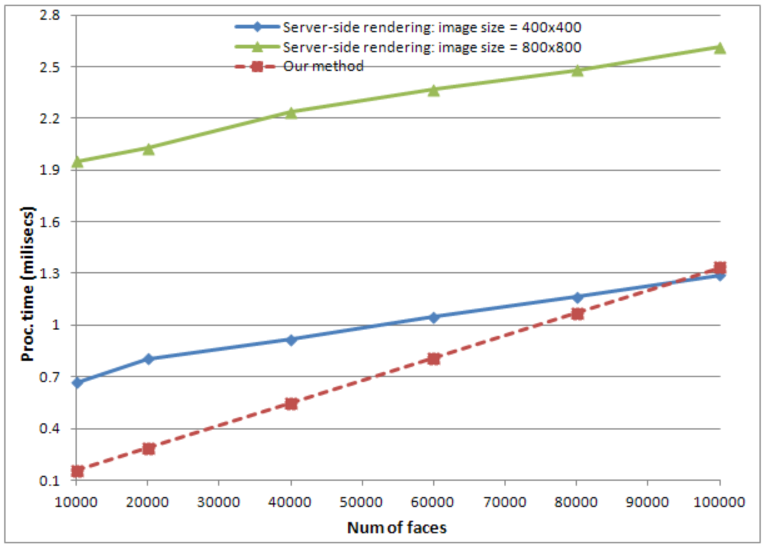


Figure 28. A comparison between server-side rendering and our method in terms of processing time tested with happy model

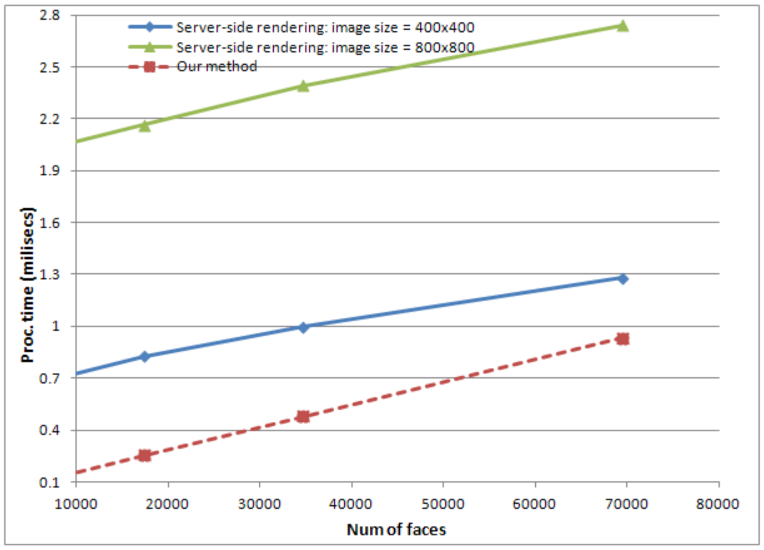


Figure 29. A comparison between server-side rendering and our method in terms of processing time tested with bunny model

b. Storage requirements

As back-face culling is performed at the server, the number of faces to be handled at the client is significantly reduced. As can be seen in the Figure below, about 40-50% of the faces are actually processed at the client. As such, our method would be of great benefits to thin clients since they are limited in their storage capacity.

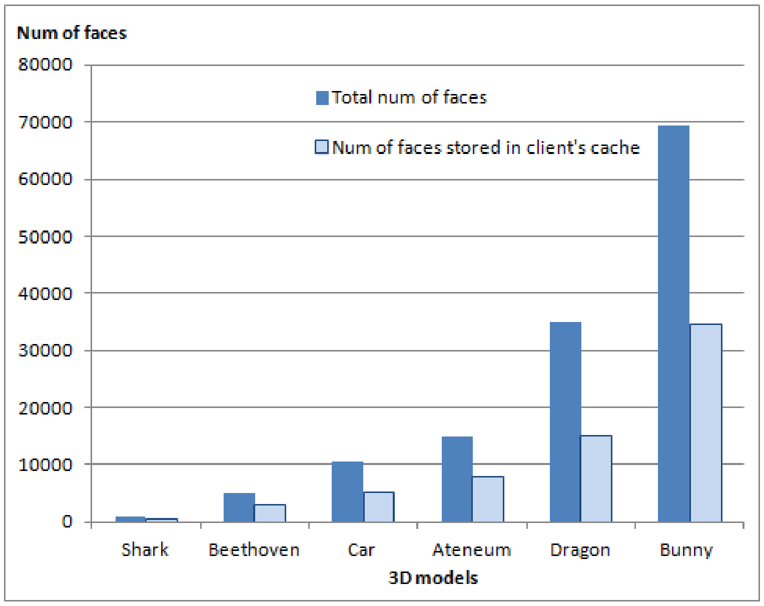


Figure 30. Average number of faces processed at the client

c. Network communication

The data transfer capabilities is considered to be the major bottleneck in the remote rendering. Network communication for the proposed framework is built on TCP/IP sockets. We employ UDP for the transmission of graphics datasets and TCP for sending commands from client to server and vice versa. We have previously presented a theoretical analysis of transmission latency in section 3.3.2. Therefore, this experiment is also able to verify the theoretical analysis of our proposed framework. Our test is conducted in both a 10 Mbps and 100 Mbps Ethernet connections.

To further reduce the transmission latency, we can make use of a compression/decompression technique. However, it is worth noting that the process of compression/decompression may also introduce some delays to the system.

Table 8. Transmission latency

Model	Num of verts	Num of faces	Latency (seconds)	
			10 Mbps	100 Mbps
Shark	468	734	0.0380	0.0043
Apple	867	1704	0.0750	0.0084
Ant	468	912	0.0380	0.0044
Beethoven	2521	5030	0.1778	0.0199
Car	5247	10474	0.3432	0.0337
Ateneam	7546	15014	0.3840	0.0469
Big dodge	8477	16646	0.5261	0.0543
Dragon 1	10006	20000	0.6247	0.0641
Dragon 2	12509	24999	0.7673	0.0802
Dragon 3	15014	30000	0.9296	0.0956
Dragon 4	17517	35000	1.0741	0.1117
Venus	19847	43357	1.2881	0.1359
Bunny	34834	69451	2.1737	0.2124

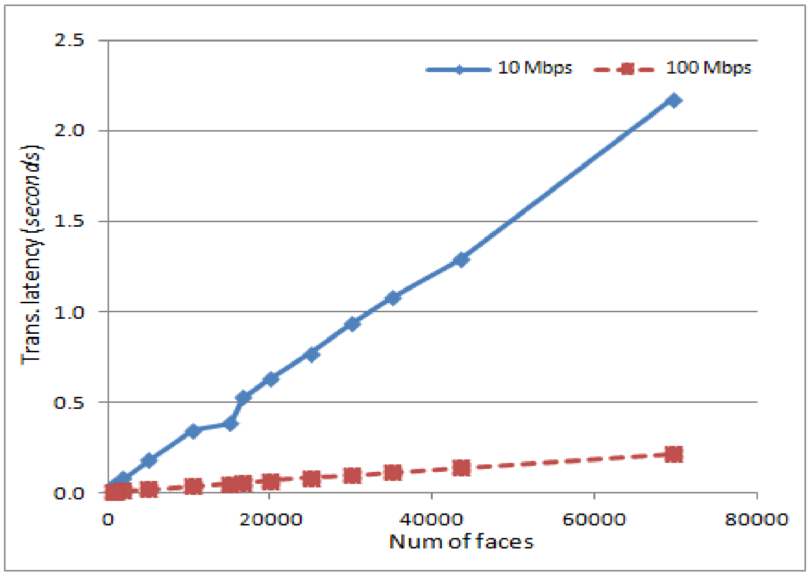


Figure 31. Transmission latency

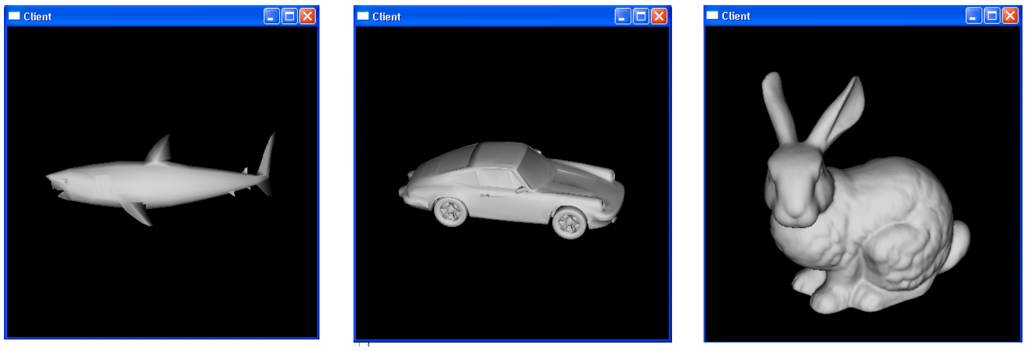


Figure 32. Client-side viewer

3.5. Summary and conclusion

In this chapter, we have investigated the graphics rendering pipeline in terms of processing time. A novel pipeline-splitting method is presented with the aim of splitting the renderings workload between the server and the client. An advantage of our method is that it can achieve hardware-acceleration on both geometry processing and rasterization stage. We have also proposed a networked rendering paradigm based on our pipeline-splitting method to facilitate remote rendering on thin clients.

Experimental results shown that our method can reduce memory cost and computational workload at the client compared to that of client-side rendering method and processing time at the server compared to that of server-side rendering method. The work also can be applied to distributed-rendering as we distribute geometry processing and rasterization to be handled on different machines in the cloud. However, the method faces a challenge to meet real time requirement due to the transmission latency. To overcome this challenge, a number of techniques can be considered to employ in order to reduce amount of data to be sent over the network such as mesh simplification and mesh compression. Additionally, in our proposed paradigm, we can see that a majority of the transmitted data between consecutive frames is likely to be redundant. In the next chapter, we will exploit this fact to propose a method which can reduce the amount of data to be sent per request therefore reduce the transmission latency.

Chapter 4

A visibility-based streaming framework for networked graphics

Chapter contents

4.1. Introduction

Interactive network-systems based on client/server architecture are posing new challenges to computer graphics. Large 3D models consisting of millions of primitives are challenging to store and render. Additionally, the transmission of large graphics datasets is considered to be a critical bottleneck in networked graphics applications. To reduce the waiting time and the amount of data being processed at the client, it is desirable to transmit only visible portions of the model to the client with respect to client's current viewpoint.

There is a significant body of work that delves into visibility-based streaming, and most are based on determination of primitives which are potentially visible from the client's current viewpoint [79-80]. The determination of visible primitives needs the collaboration of both the client and server. There are two methods for client/server collaboration. In the first one, the server is fully responsible for determining visible primitives that need to compensate the client for a proper rendering. In the second method, the client determines objects to be requested and the server sends these objects back to client on demand. These two methods respectively have great impacts on the server and the network connection workload.

In the previous chapter, we have introduced a remote rendering framework based on our pipeline-splitting method. The method can save the client in terms of processing time and computational workload. However, the number of primitives transmitted across the network may be still very high after back-face culling leading to high transmission latency. Therefore, this method is not suitable for latency-sensitive applications. In this chapter, we take a different approach for 3D mesh streaming based on server-side processing of visibility information. The pipeline-splitting and back-face culling methods are used for the determination of visible primitives. The server keeps track of primitives currently stored in the client's cache and transmits only visible primitives which are new to the client in order to reduce the number of primitives transmitted across the network. To deal with the computational workload at the server, we also present a parallel framework to scale the server side so that the computational workload can be processed in different machines in parallel.

4.3. Visibility-based streaming method

4.3.1. A theoretical analysis for visibility streaming

As the network bandwidth and transmission latency have become a critical bottleneck for interactive graphics, the back-face culling method can be of great use to the network transmission [23, 79, 81]. However, a slight change in the viewpoint might lead to a considerable number of new primitives that the network is unlikely to afford in real time. The visibility streaming method takes into account the client's cache and transmits only additional primitives that are not stored in the client's for the rendering of the next frame. Therefore, the amount of data to be sent is significantly reduced. In this chapter, we present a theoratical analysis for visibility streaming, in which a number of factors is examined, such as number of primitives needs to be sent to the client for the rendering of the next frame, the corresponding latency, and bandwidth requirement.

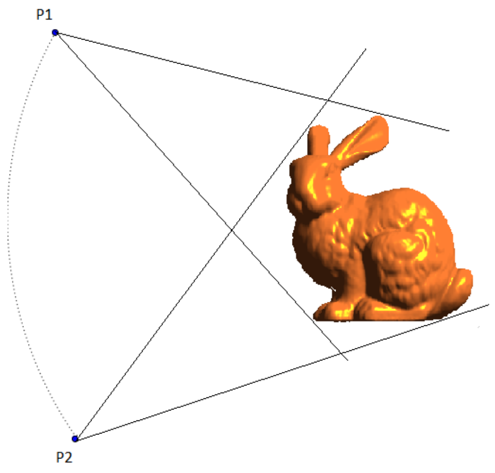


Figure 33. A movement of the camera from viewpoint P1 to viewpoint P2

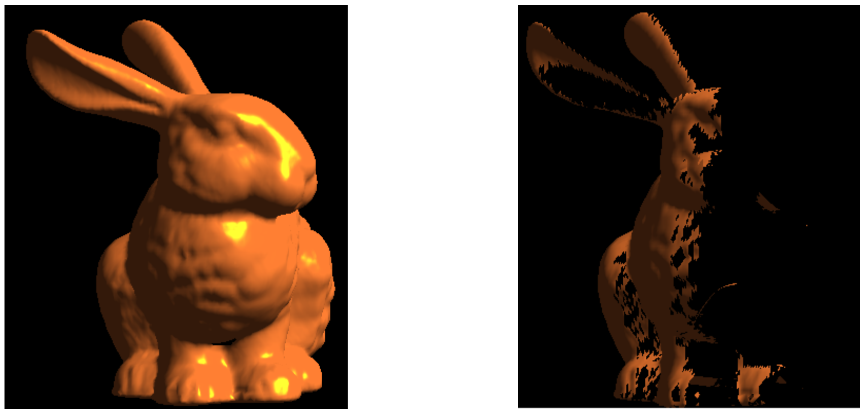


Figure 34. (a) – The model captured from the current viewpoint, - (b) New primitives appear from the movement

For the sake of simplicity, we consider an ideal case in which the 3D object has a spherical shape with radius r, composed of N_f faces. The camera is assumed to move around a concentric sphere with radious R ($R > r$).

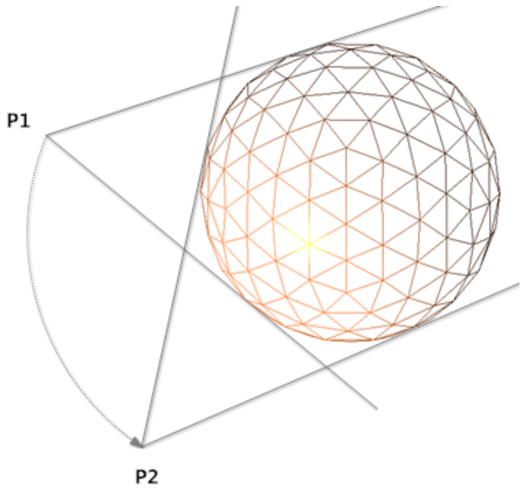


Figure 35. Spherical-shape object with radius r, composed of N_f faces

The general problems can also be considered by covering the object with a sphere then projecting all the primitives of the object into a sphere. In this case, the distribution of the primitives across the sphere is non-homogeneous as the shape of the object is no longer spherical.

We also assume that the viewing frustum is sufficient to cover the entire object. To estimate the number of faces the server needs to compensate for the movement of the camera from P_1 to P_2, we consider the plane formed by P_1, P_2 and O intersected the two spheres presented in the following figure.

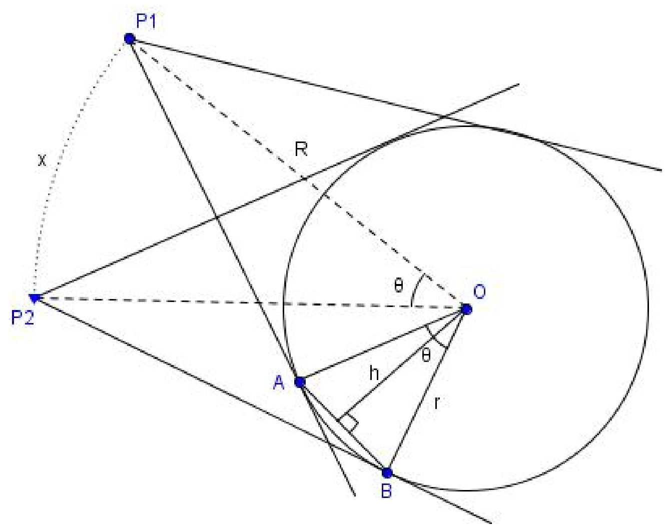


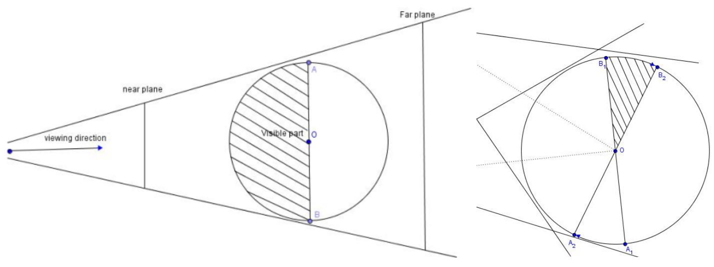
Figure 36. The intersection between the plane formed by three points (P1, P2, O) and the two concentric spheres

Table 9. Notation 4

Symbols	**Quantity**
N_f	Total number of faces of the mesh
n	Total number of faces to be sent to the client
S_{sphere}	The area of the entire sphere (the mesh)
$s_{intersection}$	The area of the spherical spherical cap where new primitives lie on

We consider an ideal case: the 3D object is a sphere with radius r and is composed of Nf faces uniformly distributed on the sphere (see Figure 13). The movement of camera (from P1 to P2) is assumed to be around a concentric sphere with radius R (R > r). We presume that the client has all information for the viewing of the camera at P1 . We also assume that the viewing frustum is sufficient to cover the entire visible part of the 3D object. Note that all visible faces are lying on a half of the sphere (Figure 13). New primitives to be sent can be calculated as follows:

$$n = \frac{s_{intersection}}{s_{sphere}} \times N_f = \frac{N_f}{2\pi}\varphi \qquad \text{(Equation 7)}$$



(a) (b)

Figure 37: (a) Viewing frustum, (b) The movement of the viewing camera

Denotes s_f as the size of a face, and bw is the network bandwidth. The total number of faces to be sent to the client for the rendering of the next frame is n. From 1 we have:

$$t_{trans} = \frac{n \times s_f}{bw} = \frac{ns_f}{2\pi bw}\varphi \qquad \text{(Equation 8)}$$

Suppose FPS is the frame rate per second that we expect to achieve. Similarly, we can calculate the requirement of the bandwidth as follows:

$$bw \geq \frac{\varphi}{2\pi} \times \frac{N_f s_f}{1/FPS - t_c} \qquad \text{(Equation 9)}$$

To reduce transmission latency, we take into account two cases of caching according to how data will be stored at the client:

Total caching: the client stores all information received from the server in its local cache. The cache will be updated after every move of the camera. This can reduce the transmission latency as there is no need to download primitives that have been received in previous viewpoints. As soon as the camera has gone through all possible positions there will be no more information to be transmitted from the server and the transmission latency ends up being zero. We consider the movement of the camera as follows:

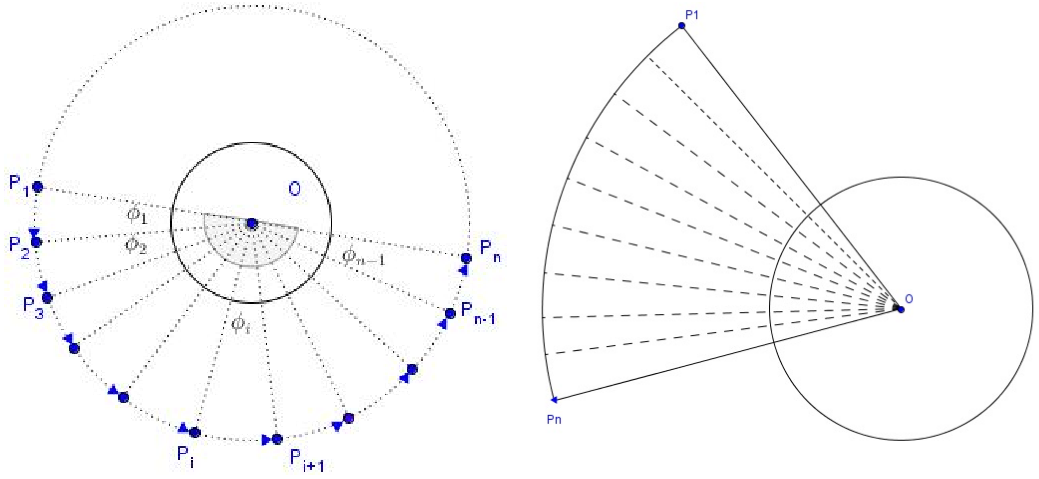


Figure 38. (a) The movement of camera, (b) slow movement of the camera

It is worth noting that to get the client cache updated with all the data from the server, the camera must undergo a number of positions from P_1 to P_2. When the camera approaches Pn all data has been updated in the cache. Let ti be the transmission latency for the movement from P_i to P_{i+1}. So the total transmission latency for the movement from P_1 to P_2 is $\sum_{i=1}^{n-1} t_i$ (Note that we do not take into account the initial time the client takes to download data at the viewpoint P_1). After this (the movement from P_1 to P_2), the transmission latency for the next movement becomes significantly reduced as most of the data has been downloaded and stored in the cache. There is possibly the case that the camera will not complete the journey from P_1 to P_2. Therefore the number of faces stored in the cache will be far less than the number of faces of the 3D model. Assuming that the camera completes its journey at P_k, let n_i be the number of faces to be sent for the movement of the camera from P_i to P_{i+1}, we can calculate the number of faces stored in the client cache (denoted as N) as follows:

$$N = \frac{N_f}{2} + \sum_{i=1}^{k-1} n_i = \frac{N_f}{2} + \frac{N_f}{2\pi}\sum_{i=1}^{k-1}\theta_i \qquad \text{(Equation 10)}$$

Let's $\widehat{P_1OP_n} = \theta$, we now can calculate N as follows:

$$N = \frac{N_f}{2}\left(1 + \frac{\theta}{\pi}\right) \qquad \text{Equation (11)}$$

Selective caching: the client stores only information which is necessary for the rendering at the current viewpoint. Only the previous viewpoint is taken into account for the computation of additional primitives to be sent to the client. Therefore, for prolonged interactions with the object, significant amount of data is needed to be sent to the client for the rendering of the next frame. This will result in the long transmission latency, but it can reduce the processing time as the number of primitives being processed at the client has been reduced.

The server is responsible for computing the list of primitives to be sent to the client for the rendering of next frame and list of primitives to be removed from the client cache. Let c_k be the list primitives remaining after culling (corresponding with the frame k) at the server side, rm_k be the list of primitives to be removed from the client cache, and r_k be the list of additional primitives to be sent to the client for the rendering of frame k. rm_{k+1} and r_{k+1} can be calculated as follows:

$$rm_{k+1} = c_k - (c_k \cap c_{k+1}) \qquad \text{(Equation 12)}$$

$$rm_{k+1} = c_{k+1} - (c_k \cap c_{k+1}) \qquad \text{(Equation 13)}$$

The client has a cache that stores all primitives which are used for the rendering of the last frame (frame k). As soon as the client receive rm_{k+1} and r_{k+1}, it needs to calculate the display list including primitives for the rendering of the next frame based on the data stored in the cache and the data received from the server. Based on the earlier calculation, we can determine the number of faces stored in the cache in case of selective caching corresponding with the angle θ_i as follows:

$$N = \frac{N_f}{2}\left(1 + \frac{\theta_i}{\pi}\right) \qquad \text{(Equation 14)}$$

4.3.2. Visibility-based framework for mesh streaming

In our proposed method, the server is responsible for computing the display list which consists of all visible primitives at the current viewpoint according to the viewing parameters received from client. The server itself has a map (indices of primitives) of the display list that is currently stored in the client's cache. It then computes a residual list containing visible primitives which are new to the client. The residual list is sent to the client for the rendering of the next frame.

To compute the residual list at the server, we make use of transform feedback mode. The server only performs geometry processing without actually having to render the 3D model by disabling the rasterization stage. The transformed primitives can be obtained at this mid-stage through the transform feedback buffer. A back-face culling algorithm (see Listing 1) is employed to cull away invisible primitives. To reduce transmission latency, only visible primitives which are not stored in the client are selected to be sent to the client.

Listing 1. The pseudo code for back-face culling

```
Vec3 vd = viewing_direction
FOR EACH triangle IN Meshes {
        Vec3 p1 = triangle.point[0]
        Vec3 p2 = triangle.point[1]
        Vec3 p3 = triangle.point[2]

        Vec3 e1 = p3 – p1
        Vec3 e2 = P3 – p2

        Vec3 surfaceNormal = crossProduct(e1, e2)
        float angle = dotProduct(vd, surfaceNormal)
        IF angle < 0 THEN render the triangle
        ELSE discard the triangle
}
```

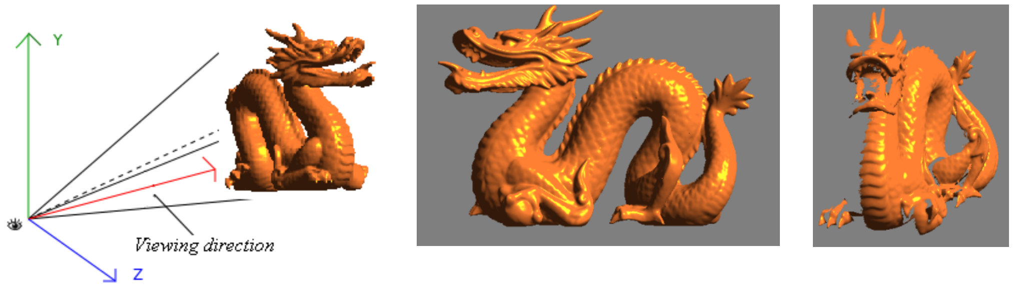


Figure 39. A back-face culling, cull all triangles faced away from the camera

The client maintains a display list which includes only visible primitives corresponding with current viewpoint. For a viewpoint change, as soon as the client receives the residual list from the server, it performs rendering with the current display list and the received residual list. The new display list then is generated by discarding invisible primitives and is stored in the cache for the next rendering.

Table 10. Notation 6

Symbols	Definitions
$F = \{f_i\}, i = \overline{0\text{-}N}$	List of faces constructed the mesh
$VF = \{f_{v_i}\}, i = \overline{0\text{-}k}$	List of visible faces
$CF = \{f_{c_i}\}, i = \overline{0\text{-}p}$	List of faces storing in the client’s cache

$RF = \{f_{r_i}\}, i = \overline{0\text{-}q}$	List of faces to be sent to the client (residual list)
$DL = \{f_{d_i}\}, i = \overline{0\text{-}m}$	Display list which is obtained at the client

The operations on the server and client are briefly summarized as follows:

Server:

1. The server performs geometry processing with the input mesh $\{f_0, f_1, ..., f_N\}$ according to the request from client to compute the list of visible faces $VF = \{f_{v_0}, f_{v_1}, ..., f_{v_k}\}$ (k<N)
2. The sever keeps track of list of faces ($CF = \{f_{c_0}, f_{c_1}, ..., f_{c_p}\}$) currently stored in the client. It then computes a residual list of faces which is in VF, but not stored in the client $RF = VF - (VF \cup CF) = \{f_{r_0}, f_{r_1}, ..., f_{r_q}\}$ $(q<k)$. RF is sent to the client for the rendering of the next frame.

Client:

1. The client renders its own data $DL = \{f_{d_0}, f_{d_1}, ..., f_{dm}\}$and is waiting for the updated data from the server
2. As soon as the client receives the residual list (RF) from the server, it renders the received data to generate the complete image of the model corresponding with the current viewpoint. The cache then will be updated with the new information based on a caching mechanism.

4.3.3. Parallel framework for visibility-based streaming

To deal with the substantial workload at the server, we propose a parallel framework for visibility mesh streaming. The computational workload at the server side can be handled in different machines in parallel.

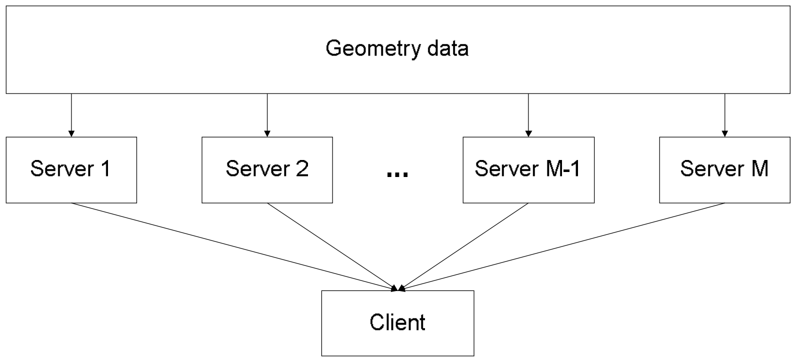


Figure 40. Parallel visibility streaming architecture

Table 11. Notation 7

Symbols	Definitions
$F = \{f_i\}$, $i=\overline{1\text{-}N}$	List of faces constructed the mesh
s_i, $i = \overline{1\text{-}M}$	List of servers
F_i, $i =\overline{1\text{-}M}$	List of primitives is handling at server S_i
VF_i, $i = \overline{1\text{-}M}$	List of visible primitives computed by server S_i
RF_i, $i = \overline{1\text{-}M}$	List of primitives computed by server S_i to be sent to the client
CF_i, $i = \overline{1\text{-}M}$	List of primitives storing in the client's cache

Assuming that the system has M servers, the mesh storing at the server side consists of N faces, $F = \{f_1, f_2, ..., f_N\}$. First, the mesh is divided into M parts: $F_1, F_2, ..., F_M$, each F_i holds a set of primitives which is part of F ($F = \bigcup_{i=1}^{M} F_i$). F_i is distributed to server s_i ($i = \overline{1 - M}$) to be handled. Each server s_i is responsible for determining list of visible faces, the so-called VF_i, from F_i ($i=\overline{1\text{-}M}$) according to the viewing parameters received from client based on the back-face culling method that we have presented earlier. Every server s_i keeps track of all primitives selected from F_i which are currently stored at the client (denotes set of those primitives as CF_i). The residual list RF_i can be easily calculated by comparing CF_i and VF_i ($RF_i = VF_i\text{-}(VF_i \cup CF_i)$), therefore only visible primitives from VF_i which are not stored in the client's cache (CF_i) are transmitted to the client to construct the new display list. As soon as the client receives all RF_i ($i=\overline{1-M}$) from the servers, it builds the display list by combining all RF_i ($i=\overline{1-M}$) with its current cache CF .

Table 12. List of models used in the test

Model name	Model index	Num of verts	Num of faces
Beethoven	1	2521	5030
Car	2	5247	10474
Ateneam	3	7546	15014
Dragon	4	15014	30000
Venues	5	19847	43357
Bunny	6	34834	69451
Horse	7	48485	96966
Blade	8	110131	220672

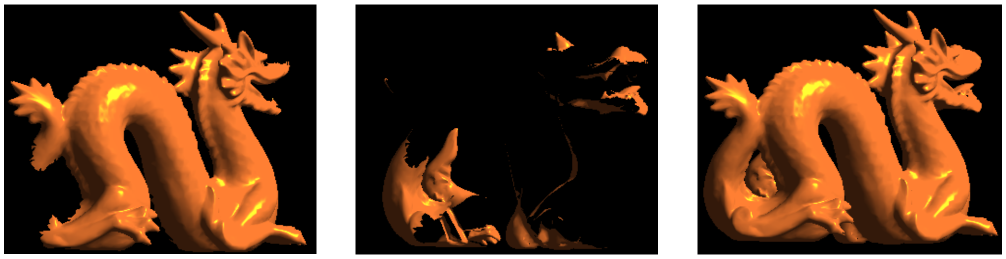


Figure 41. Parallel visibility streaming with 2 servers – dragon model: (a) image captured at the server 1 (without discarding rasterization), (b) image captured at the server 2 (without discarding rasterization), (c) image captured at the client

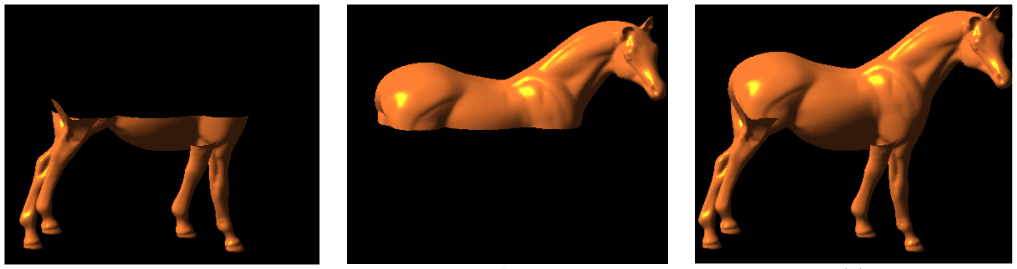


Figure 42. Parallel visibility streaming with 2 servers – horse model: (a) image captured at the server 1 (without discarding rasterization), (b) image captured at the server 2 (without discarding rasterization), (c) image captured at the client

Table 13. Parallel framework tested with two servers

Model index	**Server 1**		**Server 2**		**client**	
	Num of faces	Num of faces sent per request	Num of faces	Num of faces sent per request	Num of faces to be rendered	Num of faces received per request
1	2515	12	2515	13	3480	24
2	5237	22	5237	21	5355	44
3	7507	53	7507	53	8962	106
4	15000	130	15000	145	15929	276
5	21678	163	21678	161	21889	324
6	34725	394	34725	322	35175	716
7	48483	1661	48483	1635	54249	3296
8	110336	3322	110336	3300	117623	6622

4.4. Experimental results and discussion

We implemented a visibility-based streaming system in C++, with rendering performed through an OpenGL library. This includes the client and server modules connected via a TCP socket (or multiple TCP socket connections in the case of a parallel framework). A number of 3D models were used in the test ranging from small (which is composed of thousands of primitives) to large models (which is composed hundred thousands to millions of primitives).

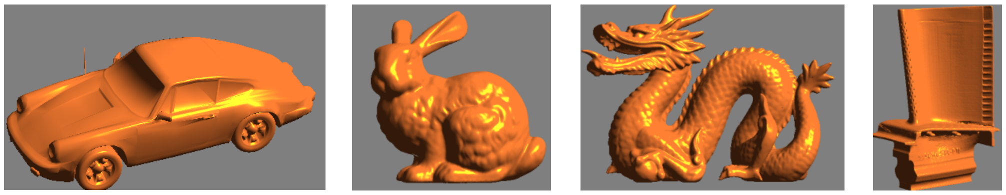
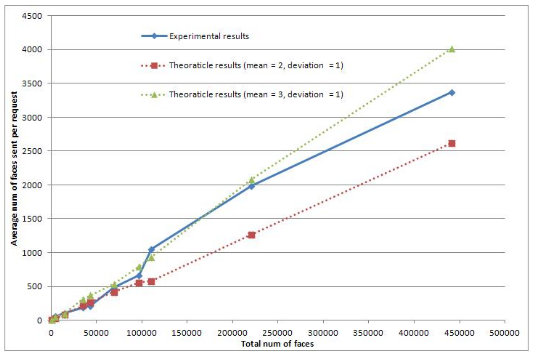
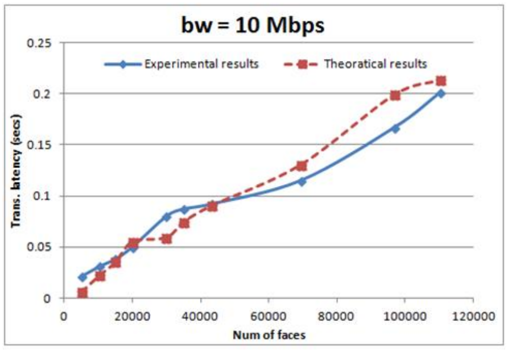


Figure 43. Several 3D models were used in the test

The residual list consisting of the number of primitives to be sent per request depends much on the complexity of the 3D models as we have previously analysed. Figure below presents the change of residual list according to the complexity of the 3D models (N_f) in terms of average number of faces to be sent per request and the transmission latency.



(a)



(b)

Figure 44. (a) Average number of faces sent per request, (b) Transmission latency

In our system, the server keeps track of primitives which are stored at the client. Therefore, the number of primitives to be sent to the client can be reduced. In addition, the number of primitives processing at the client can be significantly reduced, by up to 40-50%.

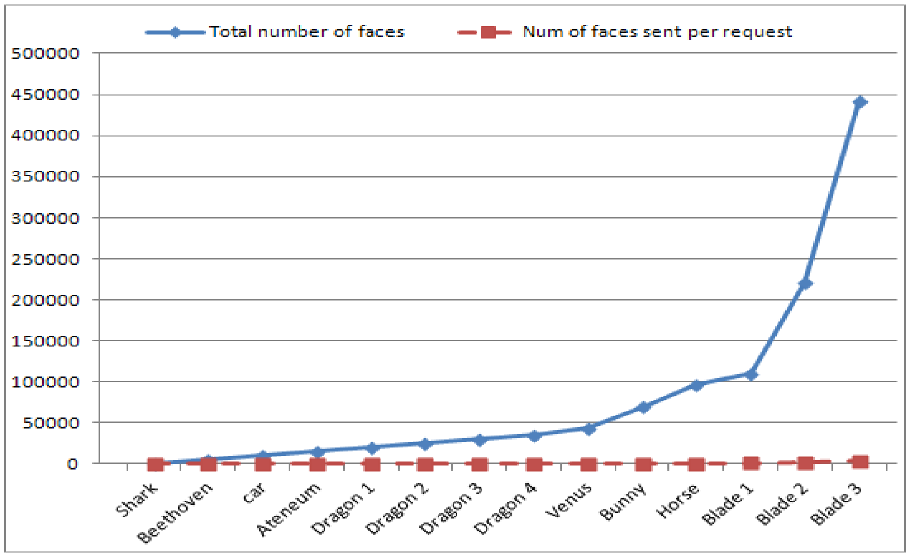


Figure 45. Number of faces to be sent per request is pretty small compared with to total number of faces of the original model

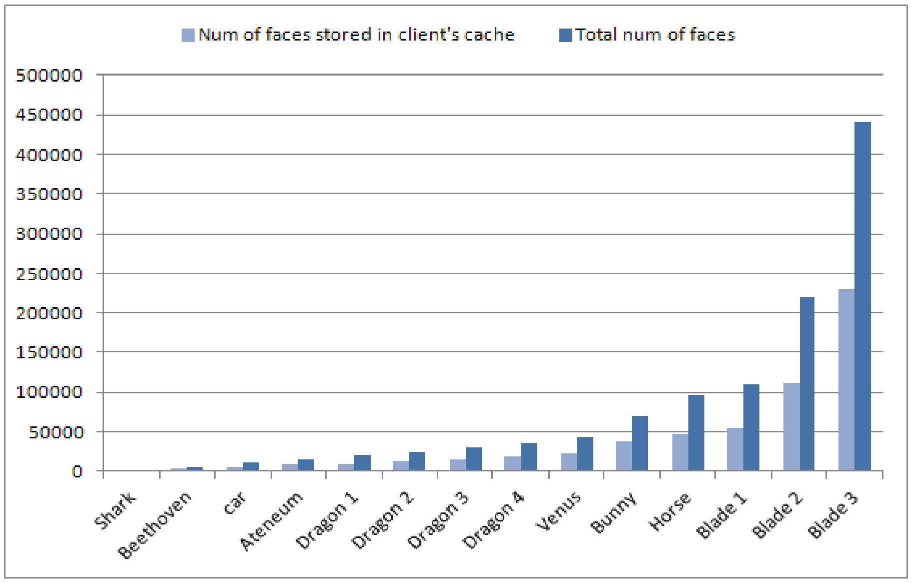


Figure 46. The number of faces processing at the client is significantly reduced

Table 14. Average number of faces processing at the client and average number of faces to be sent per request

Model	Num of verts	Num of faces	Avg. Num of faces processed at the client	Avg. Num of faces sent per request
Shark	468	734	389	5
Beethoven	2521	5030	3319	59
car	5247	10474	5611	85
Ateneum	7546	15014	9300	106
Dragon 1	10006	20000	10290	143
Dragon 2	12509	24999	12956	152
Dragon 3	15014	30000	15421	172
Dragon 4	17517	35000	18038	187
Venus	19847	43357	22588	213
Bunny	34834	69451	37791	482
Horse	48485	96966	47172	666
Blade 1	54926	110336	56505	1054
Blade 2	110131	220672	113038	1986
Blade 3	220559	441345	232789	3373

4.5. Summary and conclusion

In this chapter, we have presented a visibility-based method for 3D streaming that can effectively reduce the transmission latency. Based on a theoretical analysis, we found the relationship between the number of primitives to be sent to the client according to the viewpoint change and a number of factors such as the complexity of the 3D models and the movement of the camera. It is worth noting that our framework can work with pretty large 3D models, however, there must be a limit since the residual list is linearly proportional to the number of faces of the 3D model. In addition, the server is fully responsible for computing residual list; therefore, we also proposed a parallel framework for visibility-based streaming to scale the computational workload at the server side and to serve a large number of concurrent connections from clients.

Chapter 5

Conclusion

Chapter contents

5.1. Thesis summary

The thesis has been presented in several chapters. We can summarize our work as follows:

- We reviewed state-of-the-art approaches of remote rendering, 3D networked graphics, and graphics streaming. The trade-offs of methods, techniques are also addressed and discussed.
- We introduced a new networked rendering paradigm for remote rendering. A novel method to split up the rendering pipeline is presented, aiming to break the rendering workload from the point that geometry processing is performed at the server, leaving the remaining parts to be done at the client.

- We presented an implementation to illustrate our networked-rendering paradigm. Experimental results showed that our method can undoubtedly reduce memory cost and computational workload at the client while simultaneously yielding high quality images. However, the transmission latency is considered a critical bottleneck in our system. For complex 3D models and low bandwidth network connections, it takes a considerable amount of time to transmit graphics datasets for the rendering of each frame, thus leading to poor performance.
- A theoretical approach of visibility streaming is presented. We introduced a visibility streaming to support the transmission of 3D models across the network. A method is proposed to select visible primitives to be sent to the client based on a transform-feedback mode. To deal with the substantial workload at the server, we presented a parallel framework that can distribute the computational workload at the server to be processed on different machines. Our method can effectively reduce memory costs and network communication overhead.

5.2. Future work

The future approaches can be summarized as follows:

- Mesh compression and mesh streaming techniques can be applied to our current approach to further reduce the transmission latency.
- A sort-middle parallel rendering can be implemented based on our pipeline-splitting method.

- We are currently employing TCP and UDP for data transmission for our remote rendering implementation. A transmission protocol can be developed to assist the transmission of graphics datasets over the network

5.3. Conclusion

In this thesis, we has presented several approaches to facilitate graphics rendering on thin clients based on cloud computing. We have introduced a networked paradigm for remote rendering based on our pipeline-splitting method. The use of this method makes it possible to split the rendering workload between the server and the client. However, the transmission latency may be high due to the large number of primitives transmitted across the network. In this regard, we have also presented a visibility-based streaming framework that can reduce the amount of data to be sent over the network as well as the number of primitives processed at the client. To deal with the computational workload at the server, a parallel framework is introduced with the aim of parallelizing the processing of the workload at the server side. This allows the system to be capable of handling a large number of concurrent connections from clients.

REFERENCES

[1] S. Livatino, "Virtual Museum of Contemporary Art," in *Artificial Reality and Telexistence, 17th International Conference on*, 2007, pp. 151-156.

[2] D. Biella, *et al.*, "Virtual Museum Exhibition Designer Using Enhanced ARCO Standard," in *Chilean Computer Science Society (SCCC), 2010 XXIX International Conference of the*, 2010, pp. 226-235.

[3] S. Fleck, *et al.*, "3DTV - Panoramic 3D Model Acquisition and its 3D Visualization on the Interactive Fogscreen," in *Image Processing, 2006 IEEE International Conference on*, 2006, pp. 2989-2992.

[4] W. Shi, *et al.*, "Scalable Support for 3D Graphics Applications in Cloud," presented at the Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing, 2010.

[5] Y. Okamoto, *et al.*, "Image-Based Network Rendering of Large Meshes for Cloud Computing," *International Journal of Computer Vision,* vol. 94, pp. 12-22, Aug 2011.

[6] D. L. Kenneth Moreland, David Koller, and Greg Humphreys, "Remote rendering for ultrascale data," *Journal of Physics: Conference Series, Volume 125, Number 012096,* 2008.

[7] H. Shun-Yun, *et al.*, "FLoD: A Framework for Peer-to-Peer 3D Streaming," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008, pp. 1373-1381.

[8] H. Shun-Yun, *et al.*, "Peer-to-Peer 3D Streaming," *Internet Computing, IEEE,* vol. 14, pp. 54-61, 2010.

[9] C. Chien-Hao, *et al.*, "Bandwidth-aware Peer-to-Peer 3D streaming," in *Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on*, 2009, pp. 1-6.

[10] Q. Dinghu, *et al.*, "Mesh simplification method based on vision feature," in *Wireless Mobile and Computing (CCWMC 2011), IET International Communication Conference on*, 2011, pp. 398-402.

[11] P. Bao and D. Gourlay, "A framework for remote rendering of 3-D scenes on limited mobile devices," *Multimedia, IEEE Transactions on,* vol. 8, pp. 382-389, 2006.

[12] X. Jiang, *et al.*, "A Parallel Framework for Interactive Rendering of Massive Complex Scenes on PCs Cluster," 2007, pp. 978-983.

[13] Z. Yanfeng, *et al.*, "Parallel Rendering for Large-Scale Crowd Based on Dynamic Feedback," in *Digital Media and Digital Content Management (DMDCM), 2011 Workshop on*, 2011, pp. 172-175.

[14] K. Moreland, *et al.*, "Sort-last parallel rendering for viewing extremely large data sets on tile displays," presented at the Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics, San Diego, California, 2001.

[15] S. Molnar, *et al.*, "A sorting classification of parallel rendering," *Computer Graphics and Applications, IEEE,* vol. 14, pp. 23-32, 1994.

[16] F. Lamberti and A. Sanna, "A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 13, pp. 247-260, 2007.

[17] G. Humphreys, *et al.*, "Chromium: A stream-processing framework for interactive rendering on clusters," *Acm Transactions on Graphics,* vol. 21, pp. 693-702, Jul 2002.

[18] B. Paul, *et al.*, "Chromium renderserver: Scalable and open remote rendering infrastructure," *Ieee Transactions on Visualization and Computer Graphics,* vol. 14, pp. 627-639, May-Jun 2008.

[19] L. Sastry, *et al.*, "Supporting Distributed Visualization Services for High Performance Science and Engineering Applications A Service Provider Perspective," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, 2009, pp. 586-590.

[20] G. Xiwei and L. Hai, "Parallel Volume Rendering Based on LOD Method," in *Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on*, 2010, pp. 1-6.

[21] J. D. Foley, *et al.*, *Computer graphics: principles and practice (2nd ed.)*: Addison-Wesley Longman Publishing Co., Inc., 1990.

[22] H. Zhang and I. Kenneth E. Hoff, "Fast backface culling using normal masks," presented at the Proceedings of the 1997 symposium on Interactive 3D graphics, Providence, Rhode Island, United States, 1997.

[23] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Commun. ACM,* vol. 19, pp. 547-554, 1976.

[24] D. Cohen-Or, *et al.*, "A survey of visibility for walkthrough applications," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 9, pp. 412-431, 2003.

[25] Y. Sung-Eui, *et al.*, "Interactive view-dependent rendering with conservative occlusion culling in complex environments," in *Visualization, 2003. VIS 2003. IEEE*, 2003, pp. 163-170.

[26] T. Engelhardt and C. Dachsbacher, "Granular visibility queries on the GPU," presented at the Proceedings of the 2009 symposium on Interactive 3D graphics and games, Boston, Massachusetts, 2009.

[27] *VirtualGL* http://www.virtualgl.org/

[28] Y. Okamoto, *et al.*, "Image-Based Network Rendering of Large Meshes for Cloud Computing," *Int. J. Comput. Vision,* vol. 94, pp. 12-22, 2011.

[29] D. D. Winter, *et al.*, "A hybrid thin-client protocol for multimedia streaming and interactive gaming applications," presented at the Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video, Newport, Rhode Island, 2006.

[30] A. Jurgelionis, *et al.*, "Platform for distributed 3D gaming," *Int. J. Comput. Games Technol.,* vol. 2009, pp. 1-15, 2009.

[31] I. Nave, *et al.*, "Games@large graphics streaming architecture," in *Consumer Electronics, 2008. ISCE 2008. IEEE International Symposium on*, 2008, pp. 1-4.

[32] P. Eisert and P. Fechteler, "Low delay streaming of computer graphics," in *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, 2008, pp. 2704-2707.

[33] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," presented at the Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Boston, MA, 2010.

[34] K. Kumar and L. Yung-Hsiang, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," *Computer,* vol. 43, pp. 51-56, 2010.

[35] G. P. Gwenola Thomas, Kadi Bouatouch, "A client-server approach to image-based rendering on mobile terminals," 2005.

[36] I. M. Martin, "Hybrid transcoding for adaptive transmission of 3D content," in *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 373-376 vol.1.

[37] J. M. Noguera, *et al.*, "Navigating large terrains using commodity mobile devices," *Computers &amp; Geosciences,* vol. 37, pp. 1218-1233, 2011.

[38] G. Jung and S. Jung, "A Streaming Engine for PC-Based 3D Network Games onto Heterogeneous Mobile Platforms," in *Technologies for E-Learning and Digital*

Entertainment. vol. 3942, Z. Pan*, et al.*, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 797-800.

[39] A. Mohr and M. Gleicher, "HijackGL: reconstructing from streams for stylized rendering," presented at the Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering, Annecy, France, 2002.

[40] G. Hesina and D. Schmalstieg, "A Network Architecture for Remote Rendering," presented at the Proceedings of the Second International Workshop on Distributed Interactive Simulation and Real-Time Applications, 1998.

[41] M. Isenburg and P. Lindstrom, "Streaming meshes," in *Visualization, 2005. VIS 05. IEEE*, 2005, pp. 231-238.

[42] H. T. Vo*, et al.*, "Streaming Simplification of Tetrahedral Meshes," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 13, pp. 145-155, 2007.

[43] X. Liu*, et al.*, "A hybrid method of image synthesis in IBR for novel viewpoints," presented at the Proceedings of the ACM symposium on Virtual reality software and technology, Seoul, Korea, 2000.

[44] Z. J. Y. Lei, D. Chen, and H. Bao, "Image-Based Walkthrough over Internet on Mobile Devices," *in Proc. GCC Workshops,* pp. 728-735, 2004.

[45] Y. a. C.-O. Mann, D., "Selective Pixel Transmission for Navigating in Remote Virtual Environments," *Eurographics '97, Volume 16, Number 3,* 1997.

[46] A. Boukerche*, et al.*, "A real-time transport protocol for image-based rendering over heterogeneous wireless networks," presented at the Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, Montral, Quebec, Canada, 2005.

[47] A. Boukerche*, et al.*, "A 3D image-based rendering technique for mobile handheld devices," in *World of Wireless, Mobile and Multimedia Networks, 2006. WoWMoM 2006. International Symposium on a*, 2006, pp. 7 pp.-331.

[48] S. M. Seitz and C. R. Dyer, "View morphing," presented at the Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996.

[49] H.-C. Huang*, et al.*, "Disparity-based view morphing\&mdash;a new technique for image-based rendering," presented at the Proceedings of the ACM symposium on Virtual reality software and technology, Taipei, Taiwan, 1998.

[50] P. Bao and D. Gourlay, "Low bandwidth remote rendering using 3D image warping," in *Visual Information Engineering, 2003. VIE 2003. International Conference on*, 2003, pp. 61-64.

[51] M. Levoy, "Polygon-assisted JPEG and MPEG compression of synthetic images," presented at the Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, 1995.

[52] J. Diepstraten*, et al.*, "Remote Line Rendering for Mobile Devices," presented at the Proceedings of the Computer Graphics International, 2004.

[53] H. Y. Shum and S. B. Kang, "A review of image-based rendering techniques," *Visual Communications and Image Processing 2000, Pts 1-3,* vol. 4067, pp. 2-13, 2000.

[54] L. McMillan and G. Bishop, "Plenoptic modeling: an image-based rendering system," presented at the Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, 1995.

[55] M. Panka*, et al.*, "Visualization of multidimensional data on distributed mobile devices using interactive video streaming techniques," in *MIPRO, 2011 Proceedings of the 34th International Convention*, 2011, pp. 246-251.

[56] G. Y. Zhang and L. Zhao, "Web-based Virtual Walkthrough of Panoramas," *Ndt: 2009 First International Conference on Networked Digital Technologies,* pp. 233-237, 2009.

[57] G. M. Cortelazzo and P. Zanuttigh, "Predictive image compression for interactive remote visualization," in *Image and Signal Processing and Analysis, 2003. ISPA 2003. Proceedings of the 3rd International Symposium on*, 2003, pp. 168-173 Vol.1.

[58] a. M. V. Zoran Constantinescu, "Adaptive compression for remote visualization," *Buletinul Universitatii Petrol - Gaze din Ploiesti, vol. LXI, No. 2/2009,,* pp. 49-58, 2009.

[59] L. Chiariglione, "MPEG and multimedia communications," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 7, pp. 5-18, 1997.

[60] D. Milovanovic and Z. Bojkovic, "MPEG-4 video transmission over Internet," in *Telecommunications in Modern Satellite, Cable and Broadcasting Services, 1999. 4th International Conference on*, 1999, pp. 309-312 vol.1.

[61] A. Puri and A. Eleftheriadis, "MPEG-4: an object-based multimedia coding standard supporting mobile applications," *Mob. Netw. Appl.,* vol. 3, pp. 5-32, 1998.

[62] Y. Noimark and D. Cohen-Or, "Streaming scenes to MPEG-4 video-enabled devices," *Computer Graphics and Applications, IEEE,* vol. 23, pp. 58-64, 2003.

[63] A. B. Liang Cheng, Renato Pajarola, and Magda El Zarki, "Real-time 3D graphics streaming using MPEG-4," p. 16, 2004.

[64] H. Hoppe, "Progressive meshes," 1996, pp. 99-108.

[65] N.-S. Lin*, et al.*, "View-dependent JPEG 2000-based mesh streaming," presented at the ACM SIGGRAPH 2006 Research posters, Boston, Massachusetts, 2006.

[66] S. Rusinkiewicz and M. Levoy, "Streaming QSplat: a viewer for networked visualization of large, dense models," presented at the Proceedings of the 2001 symposium on Interactive 3D graphics, 2001.

[67] Y. Sheng*, et al.*, "A progressive view-dependent technique for interactive 3-D mesh transmission," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 14, pp. 1249-1264, 2004.

[68] B.-O. Schneider and I. M. Martin, "An adaptive framework for 3D graphics over networks," *Computers & Graphics,* vol. 23, pp. 867-874, 1999.

[69] K. Daeyoung*, et al.*, "A distance-based compression of 3D meshes for mobile devices," *Consumer Electronics, IEEE Transactions on,* vol. 54, pp. 1398-1405, 2008.

[70] M. Jianping*, et al.*, "Mobile 3D graphics compression for progressive transmission over wireless network," in *Computer-Aided Design and Computer Graphics, 2009. CAD/Graphics '09. 11th IEEE International Conference on*, 2009, pp. 357-362.

[71] M. Isenburg*, et al.*, "Large mesh simplification using processing sequences," in *Visualization, 2003. VIS 2003. IEEE*, 2003, pp. 465-472.

[72] M. Corsini*, et al.*, "Watermarked 3-D Mesh Quality Assessment," *Trans. Multi.,* vol. 9, pp. 247-256, 2007.

[73] G. Lavou\ and \#233, "A roughness measure for 3D mesh visual masking," presented at the Proceedings of the 4th symposium on Applied perception in graphics and visualization, Tubingen, Germany, 2007.

[74] J. L. Williams and R. E. Hiromoto, "Sort-middle multi-projector immediate-mode rendering in Chromium," in *Visualization, 2005. VIS 05. IEEE*, 2005, pp. 103-110.

[75] J. L. Williams and R. E. Hiromoto, "A proposal for a sort-middle cluster rendering system," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop on*, 2003, pp. 36-38.

[76] K. S. Banerjee and E. Agu, "Remote execution for 3D graphics on mobile devices," in *Wireless Networks, Communications and Mobile Computing, 2005 International Conference on*, 2005, pp. 1154-1159 vol.2.

[77] E. Agu*, et al.*, "A middleware architecture for mobile 3D graphics," in *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, 2005, pp. 617-623.

[78] J. Hultquist, "Backface culling," in *Graphics gems*, S. G. Andrew, Ed., ed: Academic Press Professional, Inc., 1990, pp. 346-347.
[79] V. Vani*, et al.*, "3D Mesh Streaming based on Predictive Modeling," *Journal of Computer Science,* vol. 8, pp. 1123-1133.
[80] P. J. N. Soumyajit Deb, "Remotevis: Remote visualization of massive virtual environments," presented at the Proceedings of National Conference on Communication, 2004.
[81] C. Bouville*, et al.*, "Efficient compression of visibility sets," Lake Tahoe, NV, 2005, pp. 243-252.

APPENDIX I – RENDERING PIPELINE ANALYSIS

Model	**Num of verts**	**Num of tris**	**Proc. Time of the entire pipeline (Tp)**	**Proc. Time at geometry stage (Tg)**
Blade 1	800124	1599996	10.9403	8.77252
Blade 3	700080	1399999	9.57553	7.67594
Blade 5	600028	1200000	8.21207	6.57963
Blade 7	499994	1000000	6.85253	5.48452
Blade 8	399945	800000	5.49629	4.38925
Dragon 4	300077	600000	4.12693	3.28573
Happy 3	199928	400000	2.83945	2.192
Dragon 12	100144	199971	1.56762	1.10597
Dragon 13	99992	200000	1.56549	1.10554
Happy 1	99953	200000	1.50261	1.09773
Blade 14	99763	200000	1.48668	1.14384
Dragon 34	10006	180000	1.42428	0.88582
dragon	10006	159999	1.29218	0.77404
big dodge	8477	140000	1.15377	0.663107
ateneam	7546	120000	1.0134	0.55275
big atc	6906	100000	0.872652	0.439507
space station	5749	60000	0.606373	0.217334
car	5247	39999	0.479758	0.106373
street lamp	4440	20000	0.338954	0.00149783
hind	3218	20000	0.336794	0.00141132
airplane	1335	16646	0.200367	0.00139806
chopper	1066	15014	0.198095	0.00139591
shark	468	13594	0.192828	0.00139543
Dragon 1	90135	10237	0.158857	0.00139532
Dragon 5	80116	10474	0.138811	0.00188587
Dragon 9	70098	8828	0.103738	0.00139389
Dragon 13	60082	6448	0.0724985	0.0014083
Blade 1	49735	2452	0.0505488	0.00139578
Dragon 26	30033	2094	0.050346	0.0013941
Dragon 30	20020	734	0.0351187	0.00139944

APPENDIX II – SOME CODES USED FOR NETWORKED RENDERING PARADIGM

Listing 2. Declare transform feedback buffer to record vertex attributes

```
// Transform feedback buffer
glGenBuffers(1, &tfvbo);
glBindBuffer(GL_ARRAY_BUFFER, tfvbo);
glVertexPointer(3, GL_FLOAT, sizeof(point), BUFFER_OFFSET(0));
glNormalPointer(GL_FLOAT, sizeof(point), BUFFER_OFFSET(12));
glBufferData(GL_ARRAY_BUFFER, _mesh->triangles.num * 3 *

sizeof(point), 0, GL_STATIC_DRAW); // we're going to record vertex

position and vertex normal
```

Listing 3. Snipped code to declare vertex attributes to be recorded to shader programs

```
glActiveVaryingNV( shaderProgram, "vertex_position\0" );
glActiveVaryingNV( shaderProgram, "vertex_normal\0" );

// link to shader program
glLinkProgram(shaderProgram);
GLint linkOk = 0;
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &linkOk);
if (!linkOk)
{
      std::cout << "Error linking shader program" << std::endl;
}

// put the shader program into use
glUseProgram(shaderProgram);
```

Listing 4. A snipped code described how to capture vertex attributes to a transform feedback buffer

```
// transform feedback

int loc[] =
{
      glGetVaryingLocationNV(shaderProgram, "vertex_position"),
      glGetVaryingLocationNV(shaderProgram, "vertex_normal"),
};

glTransformFeedbackVaryingsNV(shaderProgram, 2, loc,
GL_INTERLEAVED_ATTRIBS_NV);
glBindBufferBaseNV(GL_TRANSFORM_FEEDBACK_BUFFER_NV, 0, tfvbo);
glBeginTransformFeedbackNV(GL_TRIANGLES);
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV, query);

glEnable(GL_RASTERIZER_DISCARD_NV); // disable rasterization

// draw model
drawVBO();

glDisable(GL_RASTERIZER_DISCARD_NV);//re-enable rasterization

glEndQuery(query);
glEndTransformFeedbackNV();
```

Listing 5. A snipped code to retrieve vertex attributes from transform feedback buffer

```
/* Obtain tri data from transform feedback buffer */
tri* tfbuffer = new tri[numofIndices];
glBindBuffer(GL_ARRAY_BUFFER, tfvbo);
tri* bufferData = new tri[numofIndices];
bufferData = (tri*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);
assert( bufferData != NULL );
memcpy(tfbuffer, bufferData, numofIndices * sizeof(tri));
glUnmapBuffer(GL_ARRAY_BUFFER);
```

APPENDIX III – SOME CODES USED FOR VISIBILITY-BASED STREAMING FRAMEWORK

1. Server

Listing 6. Snipped code to compute list of primitives to be sent to the client among visible primitives

```
/*
 * Check to see if a primitive is stored in the client’s cache
 * if the returned value is:
 *          + -1  : the primitive is not in the client’s cache
 *          + else: the primitve is already in the client’s cache
 * Notes: the index of primitive is used for calculation for the
sake of
 * simplicity
 */
int in_last_list(int index, int _mi, int _ma)
{
      int _min, _max;    /* range of primitives, the index of each
*/
                         /* primitiv lies within _min and _max
*/

      _min = _mi;
      _max = _ma - 1;

      do
      {
            if (index < last_display_list[_min])     /* not in the
range */
            {
                  return -1;
                  break;
            }
            else if (index > last_display_list[_max])/* not in the
range */
            {
                  return -1;
                  break;
            }
            else if (index == last_display_list[_min])
            {
                  return _min;
                  break;
            }
            else if (index == last_display_list[_max])
            {
                  return -2;
            }
            else
            {
                  /* narrow the range */
                  _min++;
                  _max--;
            }

      } while (_min <= _max);
      if (_min > _max)
      {
            return -1;
      }
}


/*
 * Each visible primitive is checked by using the function
in_last_list,
 * only primitives which are not stored in the client’s cache are
slected
 * to be sent to the client for the rendering of the next frame
*/
void compute_residual_list()
{
      int k = 0;
      int idx;



      int _mi = 0;
      int _ma = num_last_list;


      for (int i = 0; i < num_curr_list; i++)
      {
            idx = in_last_list(curr_display_list[i], _mi, _ma);


             /* If the primitive is not in the client’s cache */
             /* put it in the list to be sent to the client */
            if (idx == -1)
            {

                  res_display_list[k] = curr_display_list[i];

                  residual_tri_list[k] =
triArray[curr_display_list[i]];

                  k++;

            }

            else if (idx >= 0)

            {

                  _mi = idx; /* mark the min value to fasten the
process */

            }

      }


      /* num of primitives to be sent to the client */

      num_res_list = k;


      /* save curr_display_list to the cache (to be
last_display_list) */

      for (int i = 0; i < num_curr_list; i++)

      {

            last_display_list[i] = curr_display_list[i];

      }

      num_last_list = num_curr_list;

}
```

Listing 7. Snipped code to retrieve primitive data from buffer object and then perform culling to select visible primitives

```
/*

 * Primitive data can be retrieved from buffer objects by using

 * glMapBuffer/glUnmapBuffer
 * Culling method is then employed to cull away invisible
primitives

 * Compute list of primitives to be sent to the client by using
the

 * function compute_residual_list

 */



void retrive_buffer()

{

      /* Retrieve primitive data from buffer object */

      tri* tfbuffer = new tri[numofIndices];

      glBindBuffer(GL_ARRAY_BUFFER, tfvbo);

      tri* bufferData = new tri[numofIndices];

      bufferData = (tri*) glMapBuffer(GL_ARRAY_BUFFER,
GL_READ_WRITE);

      assert( bufferData != NULL );

      memcpy(tfbuffer, bufferData, numofIndices * sizeof(tri));

      glUnmapBuffer(GL_ARRAY_BUFFER);



      /* perform culling */

      tri _t;

      vec3 p1, p2, p3;          /* the three points formed the
triangle */

      vec3 n;                   /* surface normal */

      vec3 cv = vec3(0, 0, -1); /* camera vector - viewing
direction */



      float angle;



      int k = 0;
```

```
    for (int i = 0; i < numofIndices; i++)
    {
        _t = tfbuffer[i];

        p1 = vec3(_t.p[0].x, _t.p[0].y, _t.p[0].z);
        p2 = vec3(_t.p[1].x, _t.p[1].y, _t.p[1].z);
        p3 = vec3(_t.p[2].x, _t.p[2].y, _t.p[2].z);

        n = (p1 - p2) ^ (p2 - p3);

        angle = n * cv;

        if (angle < 0)
        {
            k++;
            curr_display_list[k] = i;
        }
    }

    num_curr_list = k;

    /* Compute list of primitives to be sent to the client */
    compute_residual_list();
}
```

2. Client

Listing 8. Snipped code to update primitive data at client as soon as it received data from the server

```
/*

 * Update the client’s cache with primitives received from the
server

 */



void update_buffer()

{

      for (int i = 0; i < num_res_tris; i++)

      {

            curr_tri_list[i + num_curr_tris] = res_tri_list[i];

      }



      num_curr_tris += num_res_tris;



      glDeleteBuffers(1, &vbo[1]);



      /* Put data in the buffer to be drawn */

      glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

      glVertexPointer(3, GL_FLOAT, sizeof(point),
BUFFER_OFFSET(0));

      glNormalPointer(GL_FLOAT, sizeof(point), BUFFER_OFFSET(12));

      glBufferData(GL_ARRAY_BUFFER, num_curr_tris * sizeof(tri),
curr_tri_list, GL_STATIC_DRAW);

}
```