A LOW LATENCY REMOTE RENDERING SYSTEM FOR INTERACTIVE MOBILE GRAPHICS

BY

SHU SHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair & Director of Research
Professor Klara Nahrstedt
Professor John C. Hart
Dr. Zhengyou Zhang, Microsoft Research

Abstract

Mobile devices are gradually changing people's computing behaviors. However, due to the limitations of physical size and power consumption, they are not capable of delivering a 3D graphics rendering experience comparable to traditional desktops. Many applications with intensive graphics rendering workloads are unable to run on mobile platforms directly. This issue can be addressed with the idea of remote rendering: the heavy 3D graphics rendering computation runs on a powerful server and the rendering results are transmitted to the mobile client for display. However, the simple remote rendering solution inevitably suffers from the large interaction latency caused by wireless networks, and is not acceptable for the applications having very strict latency requirements.

In this thesis, I present an advanced low-latency remote rendering system that assists mobile devices to render interactive 3D graphics in real-time. My design takes advantage of an image based rendering technique: 3D image warping, to synthesize the final display image on the client from the multiple depth image references generated on the server. This *Multi Depth Image* approach can successfully reduce the interaction latency for any user interaction that changes the rendering viewpoint and at the same time maintain high rendering quality. The success of the proposed system depends on two key components: how to select the reference viewpoints and how to compress multiple depth image references efficiently. I propose a *Reference Prediction Algorithm* for real-time reference selection and a high performance warping-assisted video coding method to compress both image and depth frames in real-time. Furthermore, I also discuss about how to evaluate the real interactive performance of the proposed system. A prototype system that runs several visualization and game applications has been implemented to evaluate and demonstrate the proposed design.

To father and mother.

Acknowledgments

I always feel like this part of the thesis can be the most difficult part because there are so many people who have helped me so much. I am just afraid that the words are simply not enough to express how grateful I feel for them.

The first two must be my advisers: Professor Roy Campbell and Professor Klara Nahrstedt. They gave me the opportunity to come to the University of Illinois and start this great five-year journey. They have invested so much in sponsoring me every semester, paying for my travel to conferences, and buying everything I asked for (even though sometimes it was just a waste of money). They pushed me forward when I felt lazy, encouraged me when I was low, inspired me when I needed advices, and always supported me for trying new ideas. It is the fortune of my life to have them as my advisers.

I also sincerely thank my other two committee members, Professor John Hart and Dr. Zhengyou Zhang for offering insightful feedbacks and constructive suggestions on my research. Thank Professor Cheng-Hsin Hsu, Dr. Mahsa Kamali, and Dr. Won Jeon for helping me with the researches that complete this thesis work.

Many thanks to my colleagues in both SRG and TEEVE research groups, particularly Dr. Wanmin Wu, Dr. Ellick Chan, Ahsan Arefin, Zixia Huang, Raoul Rivas, Wucherl Yoo, Alejandro Gutierrez, Mirko Montanari, and Abhishek Verma for helping me with my paper writing, demo implementation, and make presentations. I would also like to specially thank the friends in the same lab room who I have been spending five years together with, Dr. Ying Huang, Dr. Yan Zhang, Dr. Zeng Zheng, Dr. Yong Yang, Lu Su, Chia-Chi Lin, for days and nights that we have fought together.

I leave the last paragraph for my dear parents. They have made countless sacrifices they all these years for my pursuit in academics. None of my achievements would have been possible without their supports.

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS 05-20182 and CNS 07-20702.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
FLOPS	Floating-point Operations per Second
GPRS	General Packet Radio Service
EDGE	Enhanced Data rates for GSM Evolution
UMTS	Universal Mobile Telecommunications System
LTE	Long Term Evolution
IBR	Image Based Rendering
DOL	Distortion Over Latency
RRK	Remote Rendering Kit
GLX	OpenGL extension to the X window system
AES	Advanced Encryption Standard
VNC	Virtual Network Computing
RGS	Remote Graphics Software
RTP	Real-time Transport Protocol
AOI	Area of Interest
LOD	Level of Details
PDA	Personal Digital Assistant
LDI	Layered Depth Image
DCV	Deep Computing Visualization
PSNR	Peak Signal-to-Noise Ratio
JPEG	The Joint Photographic Experts Group
MPEG	The Moving Picture Experts Group
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

CUDA	Compute Unified Device Architecture
FPS	Frame per Second
PL	Post-rendering Latency
UL	Update Latency
IL	Interaction Latency

List of Symbols

S_i	The source 3D content. The subscript i indicates the frame number. When only one frame is referred in the context, S is used for simplicity. $\mathbf{S}$ is the set of all S_i.
S'_i	A different representation of the original 3D content frame S_i. Depending on the context, it can be a subset or a simplified version.
P_i	A point cloud representation of the original 3D content frame S_i.
E_i^v	An image based environment map (e.g., panorama) of the original 3D content frame S_i at the viewpoint v.
v	A rendering viewpoint. v^+ is also the rendering viewpoint but only used in the situation to differentiate with v. vs and vc are specifically used to denote the rendering viewpoint managed on the server and client sides, respectively.
pos	One of the three vector components that describes a rendering viewpoint. pos is the position of the capturing camera.
dir	One of the three vector components that describes a rendering viewpoint. dir is the pointing direction of the capturing camera.
up	One of the three vector components that describes a rendering viewpoint. up is the facing-up direction of the capturing camera.
v_i	There are two different meaning of v with subscripts. In Chapter 3 and 4, i indicates the distance between v_0 and v_i. In Chapter 6, v_i represents the viewpoint for rendering the i-th frame S_i.
ref_i	A reference rendering viewpoint. The subscript is used when there are multiple reference viewpoints.
V	A viewpoint set. In this thesis, three viewpoint sets V_{single}, V_{multi}, and V_{double} are defined.
R_i^v	A rendering server output frame. Depending on which remote rendering approach is taken, the output can have different representations. The subscript i indicates the frame number and the superscript v is the rendering viewpoint at which this rendering result frame is processed.
Rc_i^v	The encoding result of R_i^v. The subscript and superscript have the same meaning as above.
R'^v_i	The decoding result of Rc_i^v. The subscript and superscript have the same meaning as above.
I_i^v	A rendering result image frame. It is generated by rendering the source 3D content S_i at the rendering viewpoint v.
I'^v_i	The restored result of I_i^v after encoding and decoding. $I'^v_i = dec(enc(I_i^v))$. Unless lossless coding is used, I'^v_i is different from I_i^v.

$I_i^{v \to v^+}$	It is generated by processing R'^v_i at the viewpoint v^+. The exact process depends on which remote rendering approach is taken.
$\tilde{I}_i$	An image frame that is actually displayed on the remote rendering client. According to different remote rendering designs, $\tilde{I}_i$ can either be I'^v_i or $I_i^{v \to v^+}$.
D_i^v	A rendering result depth frame. It is generated by rendering the source 3D content S_i at the rendering viewpoint v, and extracting the pixel depth value from *z-buffer*.
$W_i^{v \to v^+}$	A warping result frame. It is generated by warping depth image $< I_i^v, D_i^v >$ to a different rendering viewpoint v^+ with the 3D image warping algorithm.
$\bigcup_k W_i^{ref_k \to v}$	The composition result of k warping result frames.
Δ_i	A difference image frame. $\Delta_i^{v \to v^+}$ specifically denotes the difference between $I_I^{v^+}$ and $W_i^{v \to v^+}$.
$I(x, y)$	The pixel in the image I with x as the vertical coordinate and y as the horizontal coordinate.
BW	The bandwidth of the network connection.
rtt	The round trip time of the network connection.
FPS	The frame rate: frame per second.
err_{disk}	The threshold of the image difference when evaluating the rendering quality.
err_{resp}	The threshold of the image difference when evaluating the response time.
$\mathbf{Q_i}$	*Rendering Quality.*
T_{resp}	The time it takes a remote rendering system to respond the viewpoint change on the client side.
$\mathbf{E(T_{reps})}$	*Average Response Time.* It is the expectation of T_{resp}.
$\mathbf{p}$	*Penalty Probability.*
M	A motion vector of the user interaction that changes rendering viewpoint.
$\mathcal{S}$	A set of all input frames for video coding.
$\mathcal{R}$	A set of all R frames. $\|\mathcal{R}\|$ denotes the number of R frames in the set.
$\mathcal{W}$	A set of all W frames. $\|\mathcal{W}\|$ denotes the number of W frames in the set.
r	The actual bit rate of the encoded video. $r_{\mathcal{S}}$ denotes the bit rate of the whole video, r_{R_I}, r_{R_D}, and r_W denote the bit rate of R frame image, R frame depth, and W frame, respectively.
req	The target bit rate set for video encoding. $req_{\mathcal{S}}$ denotes the target bit rate of the whole video, req_{R_I}, req_{R_D}, and req_W are used to configure x264 to encode image, depth, and residue.
b	b_x denotes the size of the encoded frame x.
t	In Chapter 5, t_X denotes the time of playing the frame set. $t_{\mathcal{S}}$ denotes the video time. t_{R_I}, t_{R_D}, and t_W denote the time to play the component frames. Since the frame rate is the same, $t_X \propto \|X\|$. In Chapter 6, t_x denotes the time that frame x stays on the screen.

Chapter 1

Introduction

1.1 Motivation

The recent explosion of mobile devices (e.g., smart phones, tablets, etc.) is changing people's computing behaviors. The mobile device has many advantages comparing with desktops and laptops in processing daily computing tasks (e.g., replying emails, browsing web, watching videos, etc.). First, the small size and long battery life make it convenient for people to carry everywhere and use anytime. Second, the mobile users are always connected to the Internet wherever there are cellular signals. Third, the physical sensors (e.g., accelerometers, GPS, etc) provide new methods to interact with many applications and greatly improve the experience of using mobile devices. As more and more applications have been ported to mobile device, it has a promising future to become the major gateway device of computation and Internet.

However, not all applications can be directly ported to mobile platforms and the applications with complex 3D graphics belong to this category. Figure 1.1 shows a few examples. These 3D graphics applications depend on the modern GPU to perform the rendering computation and can not be supported on the mobile

(a) (b) (c)

Figure 1.1: (a) Visualization of complex 3D models [85]; (b) The state of art 3D video game [23]; (c) 3D Tele-Immersion: rendering the 3D video of real-world objects captured by multiple depth cameras [33]

platform which does not have enough computing resources. Although the latest mobile devices also have GPU cores inside, the rendering capability of mobile GPUs falls far behind desktop GPUs. Table 1.1 compares the performance difference between the state-of-arts of both mobile and desktop GPUs[1] in the market at the time of writing this thesis. Obviously, the desktop GPU has a far better performance than the mobile GPU. More importantly, due to the restriction of physical size and power consumption, mobile GPUs may not be able toto reduce this performance gap even in the future.

Table 1.1: Comparison of Mobile GPU and Desktop GPU

	PowerVR SGX 543MP2	**AMD Radeon™ HD 6990**
APIs	DirectX 9.0, OpenGL 2.1	DirectX 11, OpenGL 4.1, etc.
Core Number	2	3072 ALU, 192 TU, 64 ROP, 256 Z-Stencil
Bus Width	64 bit	256 bit
Pixel Fill Rate	1000 Mpix/s	56.3 Gpix/s
Texture Fill Rate	890 Mtex/s	169 Gtex/s
FLOPS	19.2 GFLOPS	5.4 TFLOPS (single precision), 1.37 TFLOPS (double precision)

In addition, some graphics applications require not only a powerful GPU for rendering, but also huge network bandwidth to transmit the 3D data. For example, rendering the 3D model scanned from real-world statues [39] needs to stream millions of polygons. In a 3D tele-immersive system, like TEEVE (Tele-immersive Environments for EVErybody) [92], a Gbps level bandwidth is needed to exchange all 3D video streams[2] between different sites in real-time [91]. In these cases, network bandwidth becomes another bottleneck because the wireless networks available for most mobile devices can only provide a few Mbps of effective bandwidth.

This thesis research is motivated to find a method that allows the computation and bandwidth intensive graphics applications to run on the mobile device with limited computing and networking resources.

1.2 Research Challenges

Remote rendering is a simple but effective solution. A workstation with enough computing and networking resources is used as the rendering server (in some contexts, "server" is used for short). It renders the 3D graphics contents, extracts the result image from the frame buffer, and sends to the mobile client (in some

[1] AMD Radeon™ HD 6990 (http://www.imgtec.com/powervr/powervr-graphics.asp), one of the fastest desktop GPU is compared with PowerVR SGX 543MP (http://www.imgtec.com/powervr/powervr-graphics.asp), the GPU used in Apple iPhone 4S. Since there is no benchmark that can be used to test both desktop GPU and mobile GPU, only the numbers cited from the specification sheets are compared.

[2] Unlike the 3D stereo video that has two 2D video streams for different eyes and provides the audience with depth impression, the 3D video in this thesis is reconstructed from a bundle of multiple depth image streams that are captured by different 3D cameras. It allows the user to interactively change the rendering viewpoint. It is also called free-viewpoint video according to [79].

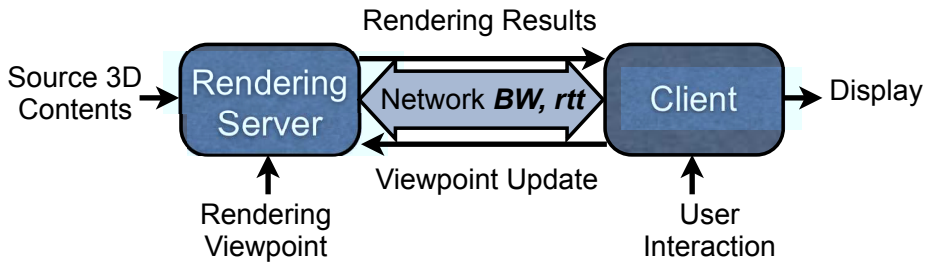


Figure 1.2: The framework of a remote rendering system

contexts, "client" is used for short). The mobile client simply displays the received 2D images. Figure 1.2 shows an illustration. In a remote rendering system, the rendering server equipped with the most advanced desktop GPUs performs 3D graphics rendering while the mobile client only receives images. Therefore, even the mobile client with no graphical hardware at all can still have the same desktop rendering experience. The network problem is also solved. The rendering server can use high-bandwidth wired network connections to receive 3D data. The bandwidth required by the mobile client only depends on the images received from the rendering server.

However, remote rendering only works perfectly for the "watch-only" applications. For the applications with frequent user interactions, it brings a new problem of *interaction latency*, which in this thesis is defined as the time from the generation of a user interaction request till the appearance of the first updated frame on the mobile client. From Figure 1.3, it indicates that the interaction latency should be no less than the network round trip time between the server and client. The long latency can significantly impair the user experience in many application scenarios. For example, the previous work [6] showed that 100 ms is the largest tolerable latency for the first person shooting games. Unfortunately, the remote rendering systems for mobile devices can have very large network latencies due to the limitation of current wireless technologies. The measurement studies [68, 45] reported 700+, 300+, and 200+ ms latencies in GPRS, EDGE, and UMTS cellular networks. Therefore, the remote rendering solution can hardly meet the latency requirement unless we manage to reduce the interaction latency less than the network latency.

Besides, there are some other challenging issues to address in order to design and build a good remote rendering system for mobile devices:

- *Compression*: A remote rendering system needs a constant bandwidth connection between the rendering server and the mobile client. Even though the bandwidth requirement only depends on the images generated on the rendering server, it still can break the maximum limit of mobile networks easily. For example, assuming the rendering images have a resolution of 640×480[3] and the rendering frame rate

[3]640×480 is a sufficient resolution to present enough details on most smartphones. However, it is not enough for most

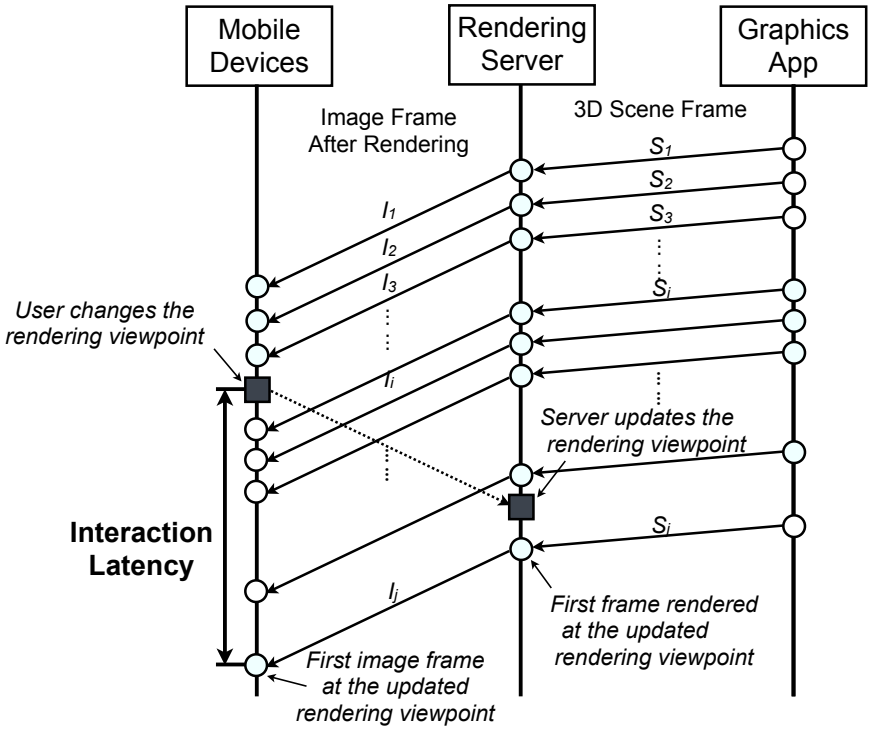


Figure 1.3: The illustration of interaction latency

is 30 fps[4], the bandwidth required to stream all uncompressed raw images to the mobile client is over 100 Mbps, which is far more than what the current mobile networks (i.e., 3G, or 4G LTE) can provide in the real world. Therefore, the rendering images have to be highly compressed before sending to mobile clients.

- *Rendering Quality*: The goal of remote rendering is to achieve the desktop level 3D graphics experience on mobile devices. Therefore, any method used to reduce network bandwidth or interaction latency should not compromise the rendering quality presented on mobile devices. For example, It is not favorable to use very large quantization parameters in video encoding because the blurred video loses the demanded rendering quality and impairs the user experience.

- *Real-Time*: For the remote rendering system with high rendering frame rate (e.g., 30 fps), the rendering server should complete the rendering and encoding computation of every frame within a deadline (e.g., 33ms). Even for the applications that do not require high rendering frame rate, the real-time processing is still demanded because any delay will be added to the *interaction latency* and affect the user experience.

tablets that have larger screens.

[4]30 fps or higher frame rate is demanded in most motion intensive applications, like games.

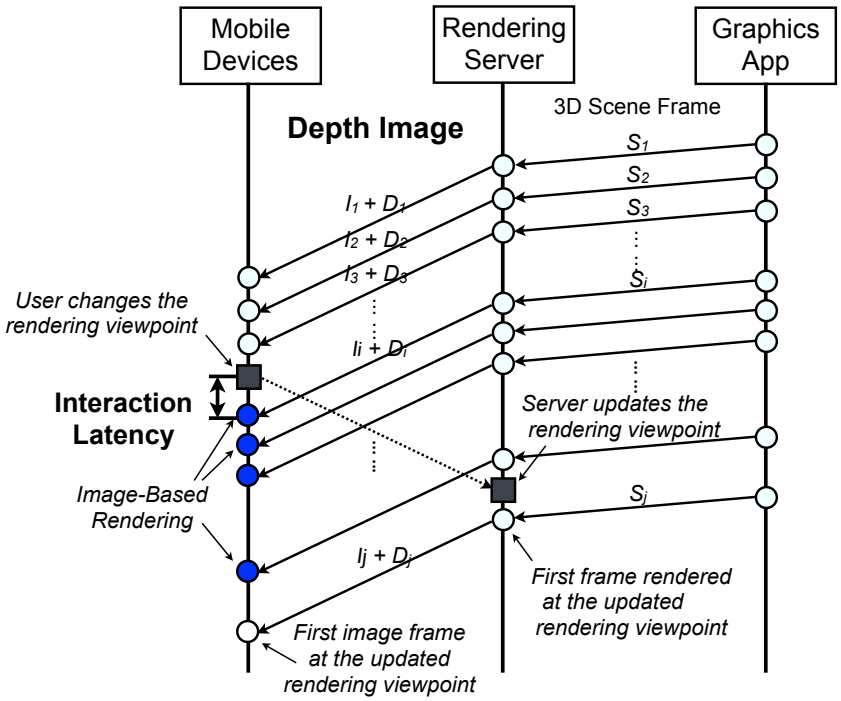


Figure 1.4: The illustration of interaction latency reduction

My thesis statement is summarized as:

> *Design and build a remote rendering system that can assist mobile devices in real-time to render complex interactive 3D graphics with high rendering quality, low interaction latency, and acceptable network bandwidth usage.*

1.3 Solution Overview and Thesis Outline

I propose a real-time remote rendering system for interactive mobile graphics. The rendering server of the proposed system sends the carefully selected depth images to the mobile client. On the client side, if there is no user interaction that changes the rendering viewpoint, the image is directly displayed. Otherwise, the mobile client runs the 3D image warping algorithm [48] to synthesize an image at the updated rendering viewpoint with the received depth images, and displays this synthesized image before any further updates are received from the rendering server. Therefore, the interaction latency of the proposed remote rendering system is reduced to the time of image synthesis on mobile, which is independent of network latency. Figure 1.4 shows an illustration.

As an outline, this thesis is organized by answering the following questions:

1. *What are other remote rendering designs proposed in the history?*

   Chapter 2 is a survey of remote rendering. Different remote rendering designs proposed in the history are summarized, reviewed, compared and analyzed. According to the type of data that is sent from server to client, remote rendering systems can be divided into two major categories: model based and image based. The model based remote rendering requires the client to share the 3D graphics rendering computation with the server. The image based systems, on the contrary, leave all the rendering computation on the server and requires no special graphical hardware on the client.

2. *What data type can be effectively used in remote rendering systems to reduce interaction latency?*

   Chapter 3 builds a remote rendering model, compares the data types used in different remote rendering designs, and concludes that using multiple depth images to represent a 3D scene frame can help reduce interaction latency. As we have introduced at the beginning of this section, the depth image can be used to synthesize the image at a different rendering viewpoint with a computation efficient IBR (image based rendering) method called 3D image warping [48]. Using two or more carefully selected depth images to warp multiple times can effectively compensate the warping artifacts and increase the quality of the synthesis image. Therefore, while the client is waiting for the server updates, the synthesis images can be displayed to reduce interaction latency (Figure 1.4).

3. *How does the rendering server generate the most appropriate depth images given the strict real-time requirement?*

   Chapter 4 formulates a reference selection problem, defines what depth images should be selected as the image based representation of the 3D scene frame, and discusses several algorithms which can be applied on the rendering server to find these depth images. The search based algorithms can find by definition the most appropriate candidates but ask for too much computation. In order to meet the real-time requirement, a reference prediction algorithm has been proposed. The algorithm predicts the reference position based on a warping error model and the evaluation indicates that the prediction result is close to the optimal result.

4. *What real-time video coding method can be applied to take advantage of the depth images and improve the compression ratio?*

   Chapter 5 introduces a novel real-time video coding method that takes advantage of the graphics contexts to improve coding performance. The performance of real-time video coding falls far behind the general-purpose offline coding because many standard optimization techniques can not be applied in the

real-time coding scenario. In the proposed remote rendering system, the encoder on the rendering server is able to utilize the camera motion information inside the graphics engine and the generated depth images to exploit more redundancies between frames. The experiments on the game play sequences indicate that the proposed real-time video coding method has the potential to beat the state-of-art x264 [24] with real-time video coding configuration.

5. *What is the appropriate method to evaluate the interactive performance of the proposed remote rendering system?*

   Chapter 6 explains the difficulty of evaluating the interactive performance of the proposed remote rendering system. Conventionally, the interactive performance is measured with latency. However, for the systems enhanced with latency reduction techniques, both latency and rendering quality should be considered in performance evaluation. A new metric DOL (distortion over latency) is defined as a weighted sum of all frame distortions during the update latency. The new metric can better evaluate the interactive performance of rendering systems with the proposed run-time performance monitor. Several real-device experiments are designed to support the conclusion.

Chapter 7 discusses some limitations of the proposed remote rendering design, suggests a few possible directions that can improve the system, and concludes the thesis.

1.4 Contributions

The contributions of this thesis can be summarized in the following aspects:

- This thesis research studies the remote rendering system from a new perspective. A network-related latency reduction problem is transformed to a content-dependent image based representation problem. Although the concepts of IBR have been studied for many years, my work presents that these old concepts can still be applied to the area of mobile computing and solve new problems.

- My research is a good example for other researchers who try to solve similar network latency problems. My design philosophy of the proposed remote rendering system is to overcome the network latency by distributing the rendering computation between the rendering server and mobile client appropriately. I believe this concept of computation distribution will be very useful for many other applications that require the integration between the cloud computing and mobile devices.

- To the best of my knowledge, my work of real-time coder is the first to study using the rendering contexts to assist real-time video coding in the area of cloud gaming, and by far, I present the best

solution that integrates the graphics rendering contexts and IBR techniques into video coding. It points out a new direction of using context information to improve the performance of the general purpose video coder in some specific application scenarios.

- The remote rendering tool RRK (Remote Rendering Kit) I have developed for the thesis research can have practical impacts on designing cloud gaming or remote visualization systems. Since I plan to publish the tool as an open source project, it will benefit other researchers who also have interests in remote rendering researches or demos.

Chapter 2

Remote Rendering Survey

The concept of remote rendering appeared early when PC was not powerful enough to process 3D graphics rendering. Sharing a dedicated graphics workstation over network to provide rendering services stimulated the research of remote rendering [56]. Besides, remote rendering has also been applied in building virtual reality or augmented reality systems where the display module (embedded wearable devices) is separated from the rendering server [2][77].

In this chapter, I will take a survey on the remote rendering systems that have been developed for different purposes in the history. The chapter is organized by dividing all cases into two major categories: general purpose design and specialized design. Typical approaches and systems of each category are reviewed in details. The survey of the first category will include the introduction of the background information of rendering technology while the discussion of the second category will present how remote rendering technologies are widely used in different areas.

2.1 General Purpose Remote Rendering

A general purpose solution is expected to support any application that follows OpenGL standard. The remote rendering function module is built into the system library. Therefore, the program can be remotely rendered without modifying the source codes.

Before introducing different remote rendering designs, it is necessary to go over some background knowledge of rendering systems. Considering most academic and open source remote rendering systems discussed in this chapter are developed on top of Linux, we use Linux as the explanation platform.

2.1.1 Rendering Basics

The 2D graphics rendering system in Unix/Linux is managed by X window system [70]. The X window system has a server-client model (Figure 2.1)[1]. Each X application runs as a client. The operating system

[1]Note that the server and client here have different definitions from those studied in remote rendering systems

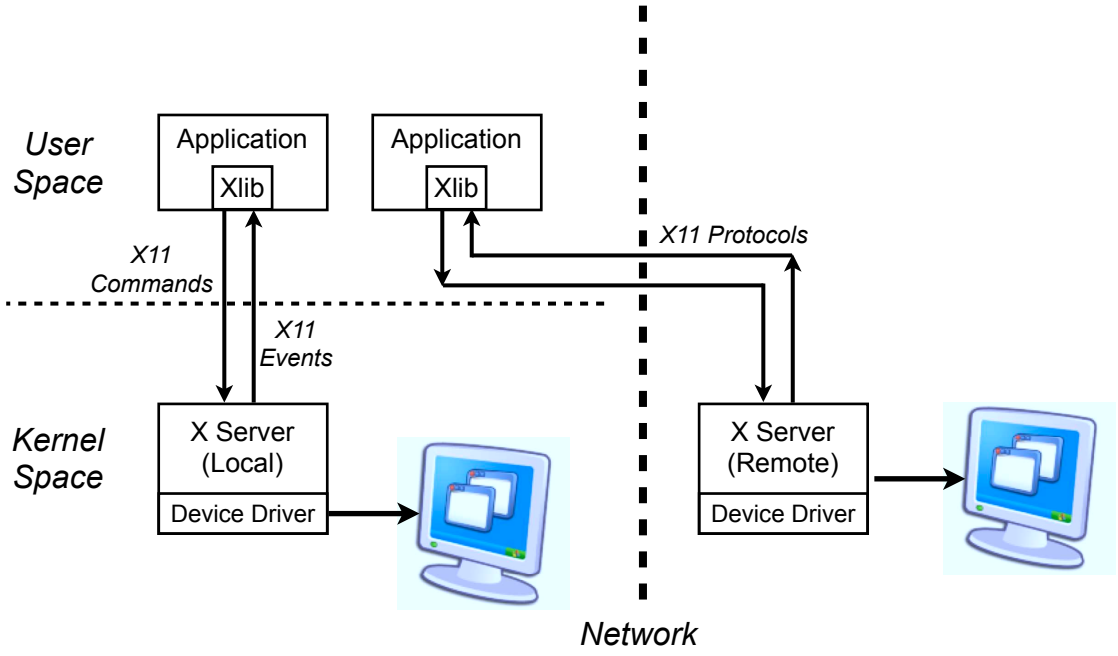


Figure 2.1: X Window Client-Server Model

hosts the X server which connects to the hardware driver. The X application (X client) generates low level XLib operations (e.g., drawing a window, filling a region with a specified color, etc.) based on what content the application wants to draw on the screen. These XLib operations are sent to X server using X protocols. The X server performs the rendering and draws the result in the display frame buffer. The design of X is network transparent, which means, X applications can display on a remote X server by sending XLib operations over network.

Compared with the 2D graphics rendering supported by native X design, the evolvement of 3D graphics rendering and OpenGL [89] makes the rendering system more complex. X server does not perform 3D graphics rendering directly. Instead, the operating system needs an OpenGL rendering pipeline, which is usually supported by graphics hardware (i.e., GPU), to process all rendering operations. 3D graphics rendering usually needs much more data than 2D rendering. The 3D data (including geometry and texture) needed for rendering a scene is not bounded by the drawing area on the display, but proportional to the size of the original 3D graphics model. It is normal to pass millions of polygons to the OpenGL pipeline when very complex scene objects are rendered. Therefore, in order to reduce the overheads of passing all 3D data through X server, the X window system allows OpenGL programs to bypass X server and directly send all rendering data (including geometry, texture, and OpenGL commands) to the OpenGL driver[2]. Figure 2.2

[2]The bypass is supported by Direct Rendering Infrastructure (DRI): http://dri.freedesktop.org/wiki/

shows an illustration.

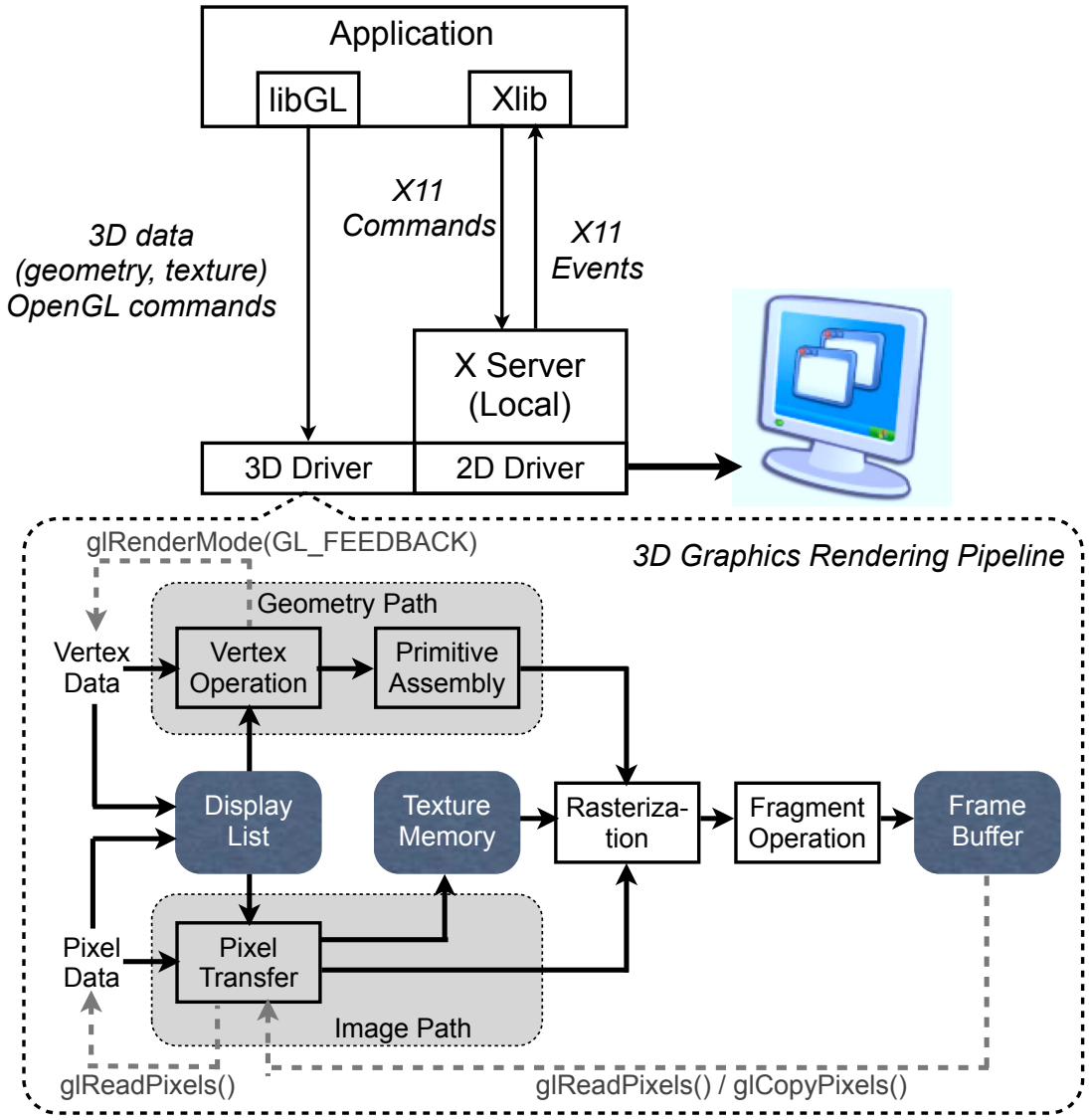


Figure 2.2: 3D Graphics Rendering with X Window System

With a big picture of the rendering system, it is easy to discover that there are three basic approaches to a remote rendering design. According to what type of data is transmitted between server and client, we can summarize these approaches as follows:

- *Model-based Approach*: 3D source data (including geometry, texture, and OpenGL drawing commands) are transmitted from server to client and 3D graphics rendering happens on the client, where X server is hosted.

- *Image-based Approach*: 3D graphics rendering happens on the server, where 3D applications run, and

only the rendering result images are sent to the client, where X server is hosted.

- *Hybrid Approach*: The data generated on the server are a mix of 3D source data and images. Both server and client share the computation of 3D graphics rendering.

2.1.2 Open Source Approaches

GLX [62] was designed as the OpenGL extension to the X window system [3]. The functions of GLX are summarized as follows:

- It comprises APIs that provide OpenGL functions to X applications;
- It extends X protocols to allow X applications (X client) to send 3D rendering data (including geometry, texture, and OpenGL commands) to the X server either on the same machine or on a remote server;
- It enables the X server that receives 3D rendering data from X clients to pass the data to the underlying 3D driver (i.e., graphical hardware).

This extension allows the 3D graphics applications to display on a remote X server like other X applications (Figure 2.3).

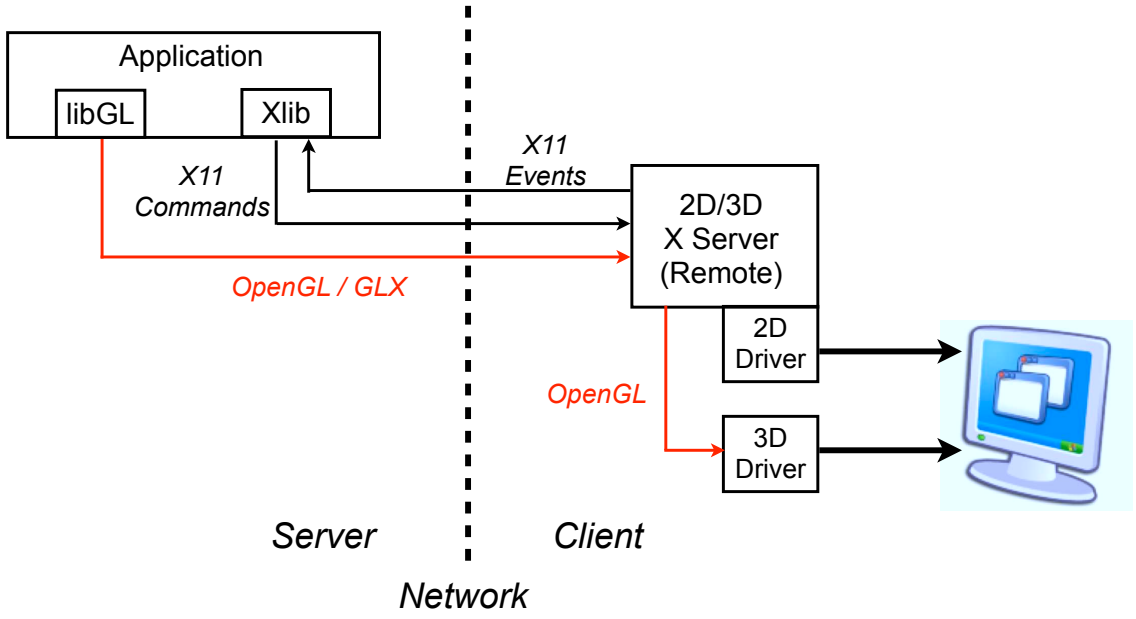


Figure 2.3: GLX in 3D Graphics Remote Rendering

NX [7] is a protocol that improves the native X protocol in the scenario of remote rendering. Compression, caching, and round trip suppression are included in the NX protocol to boost the overall speed of streaming

[3] WGL [58] and CGL[1] are similar interfaces provided for Windows and Mac OS X, respectively.

rendering operations. Although NX is not specifically designed for 3D graphics rendering, it adds the support to GLX so that the protocol can be used to accelerate 3D rendering data streaming as well.

GLX is a model-based approach. Strictly speaking, it can not be called "remote rendering" because all 3D graphics rendering computation actually happens on the "client". The "server" where 3D applications run only performs the computation to generate the 3D rendering data. This approach fits for the applications having the computation bottleneck in generating 3D data dynamically. However, it is not applicable to the mobile client scenarios because it requires the client to have powerful graphical hardwares. In addition, since all 3D rendering data is transmitted over networks, rendering complex scenes can easily overwhelm the available network bandwidth.

Stegmaier et al. integrated an image based approach into X Window system [82, 81]. For 3D graphics applications, the 3D rendering data is sent to the 3D driver on the server. After rendering, the result image is extracted, encoded, and transmitted to the client. But for other 2D graphics applications and the operating system user interface, only the XLib commands are transmitted to the client side. The client performs 2D graphics rendering and merges the received 3D rendering image into the display frame buffer. This approach also leads to an open source project: VirtualGL [13]. Figure 2.4 illustrates the framework of VirtualGL.

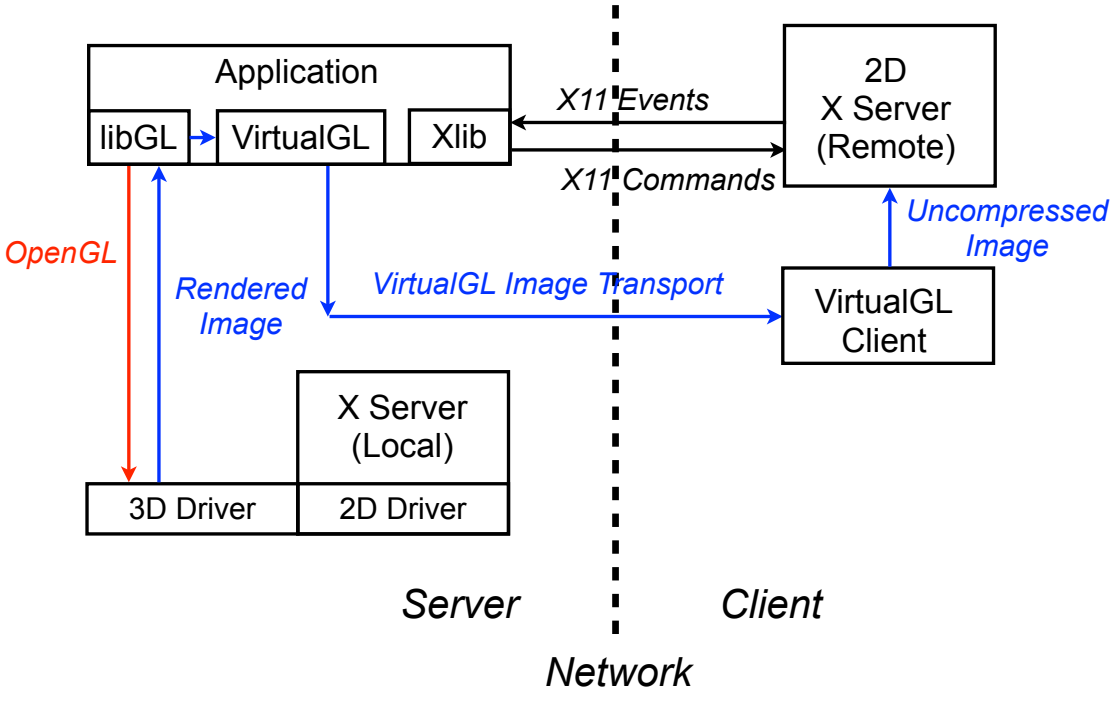


Figure 2.4: VirtualGL Framework [13]

For this image based approach, the system can take full advantage of the rendering capability on the server and requires no special graphical hardware on the client. In addition, it reduces the network bandwidth usage significantly when complex 3D models are rendered because the bandwidth for this approach only depends on the image resolution and refreshing rate.

2.1.3 Proprietary Approaches

Several proprietary general-purpose solutions have also been proposed. OpenGL Vizserver [56, 26] from SGI was very similar to the VirtualGL approach in Figure 2.4. It allowed a remote user to run hardware-accelerated graphics applications on an SGI graphics station and display on a remote client. HP Remote Graphics Software [28] also supported the hardware-accelerated remote rendering. Compared with the previous approaches, RGS featured in supporting multiple viewers to connect to a single server simultaneously for a collaborative session mode. ThinAnywhere [83] from Mercury International Technology provided similar services. A proprietary interactive Internet Protocol (iIP) was used to deliver images from server to client. The protocol secures the transportation with AES-128 encryption. In addition, the server and client of ThinAnyWhere runs on either Windows or Linux.

2.1.4 Thin Client

The thin client approaches, including VNC [67], RDP [14], SLIM [72], THiNC [4], and many others surveyed in [90, 34], have been developed to remotely access computer applications and share computing powers on the central server. For example, a VNC server maintains a software frame buffer to save all graphical outputs and sends the content of this frame buffer to a client. An open protocol, known as Remote Frame Buffer (RFB), is defined for a VNC viewer (client) to send event messages (e.g., mouse and keyboard events) to a VNC server and request screen updates from the server. THiNC took a different approach. The server simulated a pseudo video card driver to collect all low level drawing commands and sent all the commands to the client. Figure 2.5 compares the difference of thin client systems with the remote rendering approaches (X/GLX and NX) introduced previously.

Thin client is not equivalent to the remote rendering discussed in this thesis. Most early thin client systems are designed to support remote desktop which only requires 2D graphics rendering. Therefore most thin client systems do not support the remote rendering of 3D graphics natively. In addition, most thin client designs focus on the networking protocols rather than the distribution of rendering computation. Last but not the least, the pursuit of "thin" client design makes the system less capable of taking benefits from the available computing resources on the client side.

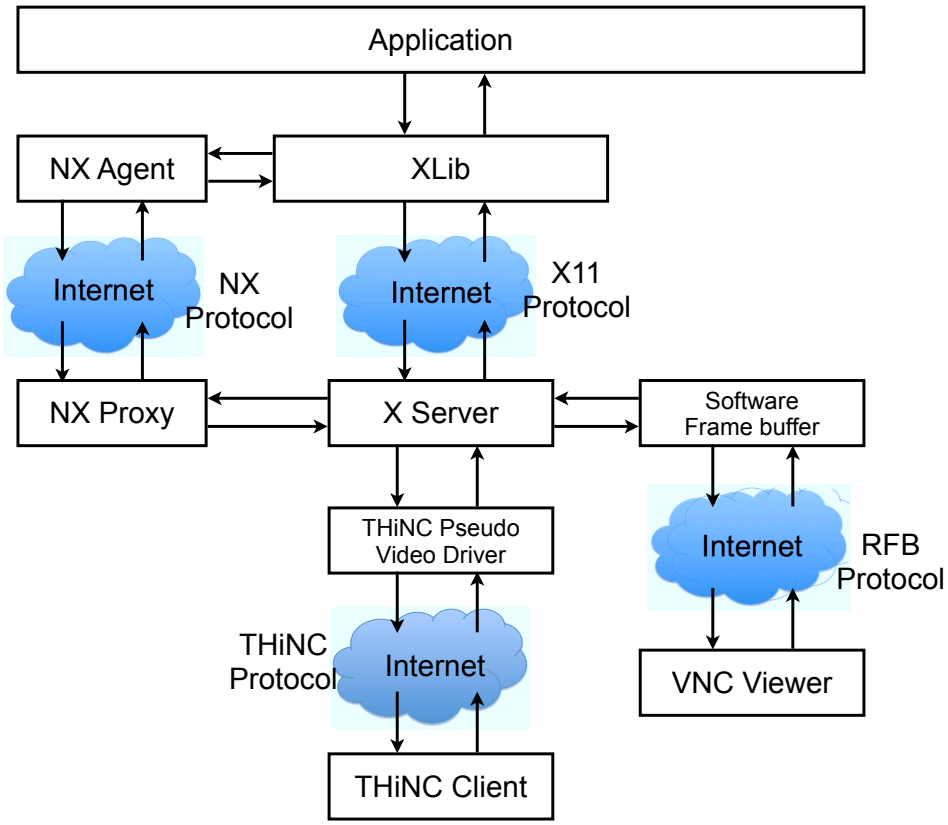


Figure 2.5: Thin Client and Remote Rendering [15]

However, it is still useful to include thin client systems in this survey. First, the later thin client designs, such as THiNC supported 3D graphics rendering and could be considered as a general purpose remote rendering system. As we have introduced earlier, the THiNC server forwards all video card drawing commands to the client and belongs to the model based streaming approach. Second, the networking protocols can be reused in remote rendering systems. For example, HP's RGS used the VNC protocol for keyboard event and screen update. VirtualGL was also integrated with VNC for network streaming. The new VNC implementation developed with VirtualGL was named as TurboVNC [13].

2.2 Specialized Remote Rendering

The specialized remote rendering solutions introduced in this section are designed from the beginning to meet the special requirements of one particular application. This section is organized based on system usage.

2.2.1 Cloud Gaming

The industry of 3D video gaming has been the major force to push the whole 3D graphics rendering technologies forward. Conventionally, the player needs to buy game consoles or the powerful GPUs to play the most advanced video games. The emerging cloud gaming service introduces the idea of remote rendering into the game industry.

OnLive[4] is a well known proprietary cloud gaming solution. It is an image-based approach: the 3D video games are rendered on the cloud server maintained by OnLive; the game scenes are extracted from the frame buffer of the rendering server and streamed to game players through broad-band networks. A customized video coder is used to compress game images and save the bandwidth usage [61]. Instead of buying any game console hardware, the game players only need to set up a broad-band network connection to play the most advanced video games freely on their PC, Mac, big screen TV (equipped with a set-up box), or even mobile devices[5]. The client needs very light hardware to receive, decode, and display the game video stream. At the same time, the client collects the user input signals (e.g., the events of mouse, keyboard, or game controller) and sends back to the rendering server to interact with game programs. The most impressive feature of this commercial system is the low interaction latency. OnLive guarantees the interaction latency for wired network users to be less than 80 ms.

There are several other proprietary solutions similar to OnLive. SteamMyGame[6] is a software that allows users to set up their own machines as the game server and stream the game to the other machine running as client. The software provider does not maintain any cloud server to provide the cloud gaming service. GaiKai[7] is a similar cloud gaming platform to OnLive but it is open to both game players and game developers. Other systems like G-Cluster[8], OTOY[9] and T5Labs[10] all claim to have the similar cloud gaming concepts.

In academia, there are also researchers working on rendering games remotely. Game@Large [18, 53] is a model-based approach. The server streams all 3D data and rendering commands to the client and the rendering actually happens on the client. Therefore, a "fat" client with graphical hardware is required. The research focused on compressing 3D data effectively to save network bandwidth and using RTP protocols for network transmission.

Jurgelionis et al. [30] proposed a hybrid approach based on Game@Large. The system can adaptively

[4] http://www.onlive.com

[5] Currently, the OnLive only allows mobile users to watch the game play because the less than 100 ms latency can not be guaranteed for wireless networks .

[6] http://streammygame.com/

[7] http://www.gaikai.com/

[8] http://www.gcluster.com/

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

[10] http://t5labs.com/

select one of the following two approaches to render games. For "fat" clients, all graphics rendering commands and 3D data are streamed as the original Game@Large design. For the "thin" client which does not have enough 3D rendering power, the games are rendered on the server and the game video compressed with H.264 [88] is streamed to the client.

Winter et al. [15] also proposed a similar hybrid approach to improve the shortcomings of traditional thin client designs in video gaming applications. The improved thin-client protocol combines both game video streaming and FreeNX [7] together. For the scenarios with intensive motions, the video encoded by x264 [24] is used; and for the static scenes, FreeNX protocol is used to send 3D rendering data to the client.

2.2.2 Virtual Environment

The early researches on the networked virtual environment focused on using multiple machines to synchronously render the same virtual world and were stimulated by different applications, such as online gaming and virtual reality systems [77]. Compared with the cloud gaming system explained in the previous subsection, the virtual environment has two features. First, it remains static and does not change over time. Thus, a virtual environment rendering system does not require a high refreshing rate. Second, the 3D graphics model of a virtual environment can be too complex to stream to one client completely in real-time for interactive rendering.

Schmalstieg's remote rendering pipeline [71] used the server as a 3D model database. The concept of AOI (area of interest) was proposed to transmit the models only viewable to client based on the current client viewpoint. In addition, the server database organizes all 3D graphic models as different LOD (level of details) and stored in Oct-Tree structure. For different zone of AOI, different level of details are transmitted. Although all the rendering computation happens on the client side, this approach allows the server to progressively transmit the complex virtual environment to the client using constrained network bandwidth.

Video streaming has also been applied for virtual environment design. An MPEG-4 streaming based virtual environment walk-through was introduced by Noimark et al. in [54]. The 3D scenes of the virtual environment rendered on the server are encoded in real-time to MPEG-4 video and streamed to the client. This approach does not need to analyze the source 3D models to determine LOD but requires constant network bandwidth between server and client for video streaming all the time.

Bao and Gourlay [3] designed a new virtual environment walk-through system using 3D image warping [48]. 3D image warping is a well-known IBR technique. Given the pixel depth and rendering viewpoint, the algorithm can efficiently warp an image to any new viewpoint. In their system, the server renders the virtual scene and streams the result image together with the depth map to the client. The client simply

displays the received images. When the user interaction changes the rendering viewpoint, the client can run the 3D image warping algorithm with both color and depth maps to synthesize the new image at the updated viewpoint. The server only needs to send the different image between the actual rendering result and the warping result to the client, which can save network bandwidth usage compared with the approach that transmits the whole image. Chang and Ger [9] proposed a similar idea to render 3D graphics scenes on mobile devices. Instead of using only one depth image, they designed the server to render the scene from different viewpoints and generate LDI (layered depth image) [75]. With LDI, the client can synthesize high quality display images without further updates from the server.

Another IBR assisted virtual environment system creates an environment map representation (e.g., pamorama). The idea originated from the famous QuickTime VR system developed by Chen in [10]. The panorama created on the server allows the client to pan in a 360-degree range without requesting further updates from server. Boukerche and Pazzi [8] presented their mobile virtual environment navigation system based on panorama rendering.

A virtual environment navigation system that adaptively selects meshes, simplified meshes or IBR representations for streaming was introduced in [73, 46, 47]. The system goal was to determine the most appropriate representation based on the client rendering capability, network performance, and user requirements.

2.2.3 Remote Visualization

Most remote rendering systems are designed for remote visualization. I divide them into sub-categories based on the system features.

- *Visualization on "Thin" Clients*: In the early days when PC was not powerful enough for 3D graphics rendering, The major motivation of studying the remote rendering framework was to achieve high quality graphics experience on the computer without high performance graphical hardware. However, the client is usually more powerful than simply displaying images.

  Levoy [38] proposed a hybrid solution to take use of the limited computing resources on the client. The server generates a simplified low-quality polygon representation from original 3D models while rendering the scene at the given viewpoint. Instead of sending rendering results directly to the client, the server sends a low-quality model and the difference image between the low-quality rendering result and the original rendering result. The client renders the low-quality model and adds the difference image to recover the final display image. This design intended to improve the problem that JPEG/MPEG encoding standards did not perform well in compressing synthetic images.

- *Visualization on Mobile Devices*: The mobile device can also be considered as a "thin" client but adding more constraints. For example, the display screens of mobile devices are usually small.

  Duguet and Drettakis [17] proposed a point based rendering scheme to take advantage of the small screen size of the mobile device. The server converts the original 3D mesh model to a point cloud, and sends the point cloud to the client. The size of the point in this point cloud representation is determined by the mobile screen resolution. Therefore, the client only renders a point cloud with the constrained size no matter how complex the original 3D graphics model is.

  Lamberti [36] used a video streaming approach to accelerate graphics rendering for PDAs because the mobile device usually has dedicated hardware to decode standard video streams. All the rendering computation is performed on the server and only the result images are updated to mobile clients. More analysis about the latency issues of this simple image streaming approach and the improvements on image compression and network streaming were discussed in his successive work in [35].

- *Web-Based Visualization*: The web-based remote rendering system allows any client using a standard web browser to access the rendering service provided by the server. The limitations of such systems include the overheads of standard web communication and the lack of assistance from client.

  The Reality Server [55] from Nvidia is a good example web-based rendering service. The rendering computation runs on the Nvidia Telsa GPUs[11]. The user can interact with the photorealistic rendering images through web browsers (with appropriate plugins).

  Yoon and Neumann [94] designed a web-based system using IBR techniques. The technique referred as "ray casting" is similar to other IBR techniques introduced in view interpolation [11] and 3D image warping [48]. The performance of their system was compared to other image-based approaches using JPEG or MPEG-2 standards in terms of the rendering result compression ratio.

- *Large Scale Volume Data Visualization*: Rendering large scale volume data remotely is practically demanded in the area of medical image visualization. For example, the visualization of CT scan results is very helpful for the doctor to diagnose. However, the size of the volume data is usually too large to transmit over networks and render on less powerful computers or mobile devices.

  The interactive Micro-CT scan exploration system designed by Prohaska et al. [64] stores all the volume data in the server and retrieves the sub-volume data for the client to render. Engel et al. [20] developed a remote rendering system for very large volume data set visualization. The server renders

[11] According to http://www.mentalimages.com/products/realityserver/realityserver-software/technical-specifications.html, Quadro series GPUs are also supported when the server runs on Linux platform.

the volume to 2D slices with texture mapping and sends the requested slice to client. The client renders the 2D slice locally. This hybrid approach takes advantage of both the high-quality rendering on server and the local rendering for latency reduction. Ma and Camp [42] addressed the issues of rendering time-varying volume data for remote clients. The main contribution of their system was to utilize multi-core super computers to render the volume data in parallel.

- *Security*: Remote visualization systems are also used to protect valuable 3D source data from leaking to malicious users. Koller et al. [31] developed this idea into a protected interactive 3D graphics rendering system. The server stores the valuable 3D model data scanned from the famous artwork, and provides the rendering service to client users. Only the rendering image results are sent to the client so that the 3D sources are well protected.

2.3 Collaborative Rendering

Some rendering systems can not be strictly named as remote rendering, but very related to my thesis research. I summarize them in a separate category: collaborative rendering.

Mark [44, 43] proposed to use post-rendering technique to interpolate new frames between the image frames that are actually rendered by the 3D graphics rendering engine. His system aimed to increase the rendering frame rate because the GPU was not powerful enough at that time to render complex 3D scenes fast enough to meet the refreshing requirements. A 3D image warping engine driven by CPU was used to work together with GPU simultaneously to generate more frames. Smit et al. [78] took a very similar approach in building a virtual reality rendering system in order to increase the rendering rate to 60Hz. These are the examples of the collaboration between CPU and GPU.

Zhu et al. [95] developed a collaborative rendering system to enable light-weight users (e.g., mobile client) to join Second Life [12]. Conventionally, a Second Life client needs to download the virtual world map and all avatar's information from the game server and render the game scene locally. In Zhu's system, not all clients need to have powerful graphical hardware to rendering the virtual world. The light-weight player can request the game scenes (in the format of depth image) rendered by his neighbors[13] and synthesize the display image from the received depth images. Zhu also studied how to extract foreground objects from background scenes to achieve better quality. In this case, the rendering is collaborated with network peers.

In the situation that 3D data are too large for only one machine to finish rendering, the workload will be divided into small parts and collaborated within a cluster. IBM's Deep Computing Visualization (DCV)

[12] http://www.secondlife.com/

[13] The neighbor refers to the players whose avatars are close to this player's avatar in the virtual world

system [80] offers a combination of scalable rendering infrastructure and remote delivery of imagery in a client-server configuration. On the back end, DCV intercepts the OpenGL command stream and routes subsets of the command stream to different nodes of a distributed-memory system for rendering. Sub-images from each node are then collected and combined into a final image, which is then compressed and sent out to a remote client for display. Like VirtualGL, the DCV system sends the XLib protocol directly to the client for processing. OpenGL pixels are compressed and sent over a separate communication channel to a custom OpenGL client. The communication protocol for transporting OpenGL imagery is proprietary.

Chromium RenderServer [59, 29] is another scalable remote rendering system. The rendering server is built on the cluster where the rendering data set is huge and needs to be paralleled to different nodes. The major improvement of Chromium RenderServer over IBM DCV is the support of parallel rendering. Other than that, the system is designed to be compatible with RFB protocol used by VNC viewers so that the client of Chromium RenderServer can be any VNC viewer. Lamberti et al. [35] proposed a video streaming system to improve Chromium. The rendering cluster was used not only for 3D rendering, but also for distributed video coding to compress the MPEG-4 video in real-time.

2.4 Summary

I have briefly surveyed different remote rendering designs proposed in the history. My survey does not cover many technical details of each design, but shows a big picture of how remote rendering technologies can be applied to solve different problems.

Neither the model-based approach nor the image-based approach can be directly used for remote rendering for mobile interactive graphics as required in my thesis research. Sending 3D data does not help solve the networking and computation problems of mobile devices and using only image frames in remote rendering has the serious problem of interaction latency.

Some specialized approaches using other forms of representation may be appropriate for mobile devices. The detailed analysis of how each approach can be used for the purpose of interaction latency reduction is discussed in the next chapter.

Chapter 3

System Analysis and Design

In this chapter, I will first study how to design a remote rendering system that can stream complex 3D graphics scenes to mobile devices with low interaction latency. A remote rendering model is developed and used to analyze the previous designs surveyed in Chapter 2. Then I present the core of this thesis research, the design of a real-time low-latency remote rendering system, and demonstrate a prototype system that I have implemented.

3.1 Remote Rendering Model

This model is built to characterize the key requirements in designing a remote rendering system. $\mathbf{S} = \{S_i\}$ denotes all source 3D contents. For every scene, the server takes S_i as input, processes some operations, and generates a result R_i^{vs}:

$$R_i^{vs} = proc\_s(S_i, vs)$$

where vs is the rendering viewpoint maintained on the server. R_i^{vs} is usually encoded before network streaming.

$$Rc_i^{vs} = enc(R_i^{vs})$$

The encoded data is sent into the network, that has a bandwidth BW and a round-trip time rtt[1]. On the other side, the client runs decoding to retrieve the data.

$$R'^{vs}_i = dec(Rc_i^{vs})$$

In our research, we assume that the distortion caused by encoding/decoding is minimal.

$$diff(R_i^{vs}, R'^{vs}_i) < \epsilon$$

[1]For simplicity, we add all network related factors to rtt, such as the retransmission due to packet loss, control overheads, the delay caused by traffic congestion, the waiting time in a streaming buffer, etc.

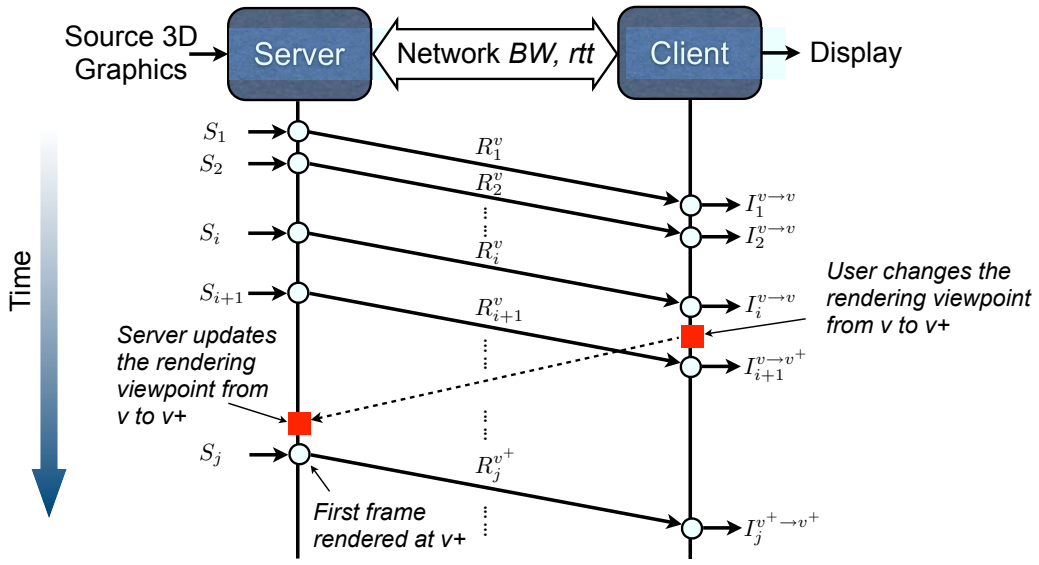


Figure 3.1: Illustration of remote rendering model

Therefore, we use R_i^{vs} and R'^{vs}_i interchangeably when encoding/decoding is not specially mentioned in the contexts. After the client has R'^{vs}_i, it generates the display image:

$$I_i^{vs\to vc} = proc\_c(R'^{vs}_i, vc)$$

where vc is the rendering viewpoint maintained on the client. For most time, vc should be equal to vs. However, as illustrated in Figure 3.1, there exists a period of viewpoint inconsistency when user interaction happens.

This simplified model can be used to analyze a remote rendering design. In Chapter 1, I have summarized four requirements for a remote rendering system:

1. *Latency*: The interaction latency should be reduced so that the system can respond to user interactions as fast as possible.

2. *Quality*: The client is expected to display a good quality rendering result, as if the client has the same computing resources as the server;

3. *Real-Time*: All rendering and encoding computation should be completed before the next rendering frame arrives.

4. *Compression*: The data that is sent from the rendering server to the client should not use more network

bandwidth than expected.

These requirements can be formally represented by equations using our remote rendering model.

First, *average response time* is used to reflect how fast the system is responsive to the user interaction that changes the rendering viewpoint (i.e., the user interaction changes the rendering viewpoint on the client from v to v^+. Before the update request is received by the server, $vs = v$, $vc = v^+$). The *response time* T_{resp} is defined as follows:

$$T_{resp}(v \rightarrow v^+) = \begin{cases} T(proc\_c) & diff(I_i^{v \rightarrow v^+}, I_i^{v^+}) < err_{resp} \\ T(proc\_c) + T(proc\_s) + T(enc) + T(dec) + rtt + \frac{\|Rc_i^{v^+}\|}{BW} & else \end{cases} \tag{3.1}$$

where the function $T(func)$ returns how long it takes to execute $func$, and $\|Rc_i^{v^+}\|$ is the frame size. When the user interaction changes the client rendering viewpoint from v to v^+, the client should update the viewpoint change to the server. At the same time, the client runs *proc_c* again based on the existing R'^v_i and generate $I_i^{v \rightarrow v^+}$. If the image $I_i^{v \rightarrow v^+}$ is already similar to the standard rendering result $I_i^{v^+}$, the *response time* is only the execute time of *proc_c*. Otherwise, the client needs wait until the updated frame $Rc_i^{v^+}$ is sent back from the server. This is the interaction latency problem discussed in Chapter 1. Based on Eq. (3.1), *average response time* is represented as $\mathbf{E}(\mathbf{T_{resp}})$:

$$\mathbf{E}(\mathbf{T_{resp}}) = T(proc\_c) + \mathbf{p} \cdot \left(T(proc\_s) + T(enc) + T(dec) + rtt + \frac{E(\|Rc)\|)}{BW} \right) \tag{3.2}$$

where $E(\|Rc)\|)$ is the average size of the data frames generated on the server. $\mathbf{p}$ is the *penalty probability* that $I_i^{v \rightarrow v^+}$ is not similar to $I_i^{v^+}$

$$\mathbf{p} = P\left(diff(I_i^{v \rightarrow v^+}, I_i^{v^+}) \geq err_{resp} \right) \tag{3.3}$$

The goal is to have the *average response time* less than the largest tolerable latency $delay_{max}$. From Eq. (3.2), the most effective method to reduce $\mathbf{E}(\mathbf{T_{resp}})$ is to reduce the *penalty probability* $\mathbf{p}$. If $\mathbf{p}$ can be reduced to zero, the interaction latency becomes independent of network (rtt). Therefore, the *latency* requirement can be presented as:

$$T(proc\_c) < delay_{max} \quad and \quad \mathbf{p} \rightarrow 0 \tag{3.4}$$

Second, the *rendering quality* metric $\mathbf{Q_i}$ is used as the quality of the images displayed on the client when

there is no viewpoint inconsistency (i.e., $vs = vc = v$). The definition of $\mathbf{Q_i}$ is:

$$\mathbf{Q_i} = \mathit{diff}(I_i^{v \to v}, I_i^v)$$

where I_i^v is the result image of rendering S_i at the viewpoint v.

$$I_i^v = render(S_i, v)$$

Please note that I_i^v may not be actually generated in the system, but only used to evaluate the *rendering quality* in this model. In order to present a good rendering quality on the client, $\mathbf{Q_i}$ is expected to be zero or at least less than a threshold. Thus the *quality* requirement is:

$$\forall i, \quad \mathbf{Q_i} < err_{disp} \tag{3.5}$$

Third, the *real-time* requirement can be formalized as:

$$T(proc\_s) < \frac{1}{FPS_s}, T(proc\_c) < \frac{1}{FPS_c} \tag{3.6}$$

For the rendering applications that dynamically generate source data and refresh screen at a high frame rate (e.g., games, 3D video, etc.), FPS_s is the frame rate of source data generation and FPS_c is the refresh rate on the client (FPS_s should be no smaller than FPS_c).

Last, the *compression* requirement is determined by the network bandwidth, the rendering frame rate, and the size of the server output frame:

$$\frac{\|Rc_i^v\|}{BW} < \frac{1}{FPS_c} \tag{3.7}$$

3.2 System Analysis

In this section, the remote rendering approaches surveyed in the previous chapter will be analyzed with the proposed remote rendering model. The focus is on the four requirements defined in Eq. (3.4), (3.5), (3.6), and (3.7). The remote rendering systems are organized according to the server output data type: model-based and image-based (Figure 3.2).

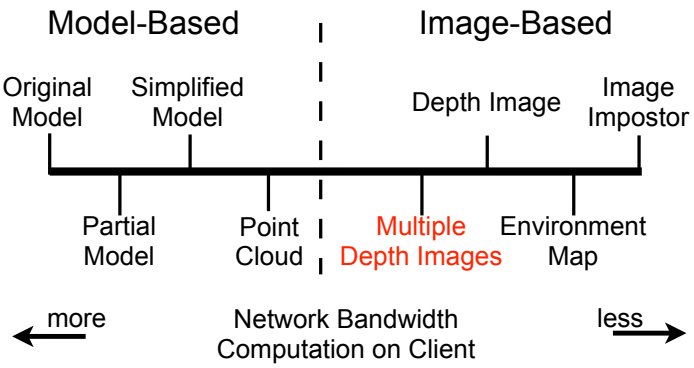


Figure 3.2: Classify remote rendering systems based on data types

3.2.1 Model-Based Remote Rendering

The model-based remote rendering systems send 3D data to the client, which is expected to have enough hardware to run 3D graphics rendering. According to granularity of the 3D models generated on the server, I can create the following sub-categories.

Original Model

GLX [62], Game@Large [18], and THiNC [4] are good examples of this approach. The rendering server simply forwards all 3D data to the client for rendering. It is useful for the application that requires heavy computation in generating rather than rendering the 3D model data[2]. In this case, $\mathbf{Q_i}$ equals to 0 if the graphics rendering pipeline on the client follows the same standard. $\mathbf{p}$ is also 0 by definition. However, the interaction latency is not actually reduced because $T(proc\_c)$ depends on the capability of the graphical hardware on the client and the complexity of 3D data. For the scenario when the client has very limited computing resources, the requirements of Eq. (3.4) and (3.6) can not be met. Moreover, streaming complex 3D models over networks is against Eq. (3.7).

Partial Model

Some remote rendering designs only transmit a subset of the original model to the client [21, 64, 71]. The rendering server of these approaches generates a subset S_i' of the original model S_i. Compared with *Original Model*, this approach can release the network contention and reduce the start time on the client side at the cost of performing extra computation on the server side to divide models into subsets at runtime. Meanwhile, $\mathbf{Q_i}$ and $\mathbf{p}$ both depend on how S_i' is created.

[2]In my remote rendering model, the computation of generating 3D data for scene rendering is excluded from $proc\_s$.

Simplified Model

The most representative design of this approach is [38]. The server generates a simplified version S'_i of the original model S_i and a difference image Δ_i. The client renders the simplified model and adds the difference image to restore the expected rendering result, so that $\mathbf{Q_i}$ is perfect. The system needs only a light-weight rendering engine on the client, requires much less network bandwidth for streaming models, and maintains rendering quality. In return for the benefits, the server is heavily loaded to analyze the original model, generate a simplified model, render both models and compare the difference. Besides, $\mathbf{p}$ depends on how S'_i is generated because Δ_i does not help when the viewpoint is changed.

Point Cloud

This is a special case of *Simplified Model*. A point cloud representation of the original model [17] P_i generated on the server with a resolution comparable to the display screen of the client and therefore can be much easier for network streaming and 3D graphics rendering. The overhead is that extra computation on the server is required to transform the mesh-based model to a point cloud. Similar to *Partial Model*, both $\mathbf{Q_i}$ and $\mathbf{p}$ depend on how P_i is created.

3.2.2 Image-Based Remote Rendering

An image-based remote rendering system renders all 3D models on the server and sends images to the client. Unlike the model-based approaches, the client does not need to have any graphical hardware for 3D graphics rendering.

Image Impostor

Most remote rendering designs are taking this simple approach. Only one 2D image per rendering frame is sent from the server to the client. Considering the image impostor only needs a resolution comparable to the display resolution on the client, the required network bandwidth for streaming images is bounded regardless of the complexity of original 3D models. Moreover, the 2D images can be efficiently compressed with the advanced image and video coding techniques to save streaming bandwidth. For *Image Imposter*, the only drawback is the highest *penalty probability*. If we assume the image rendered at different viewpoints are different,

$$diff(I_i^{vs}, I_i^{vc}) > err_{disp},\ if\ vs \neq vc$$

Then $\mathbf{p} = 1$ and every user interaction suffers a full network round trip latency.

Environment Map

Environment map has been widely used in game development to simplify the rendering of far away background objects. In the scenario of rendering virtual environment remotely [10, 8], it can be effectively applied to reduce the interaction latency. Given the position of the viewpoint, the server generates a panoramic environment map with a 360-degree coverage of the whole world. On the client side, the environment map can be projected to the standard view image at any view direction, which allows the user to freely pan the virtual camera without further rendering assistance from the server. Therefore, the latency for any panning motion only depends on the time of running view projection on the client. But for other user actions (e.g., change the position of virtual camera), the client still needs to wait for the server to send new environment maps. The *penalty probability* can be written as $\mathbf{p} = P(vc.pos \neq vs.pos)$.

Depth Image

In this approach, the server renders the 3D model and generates a depth image (with both color map and depth map). When the client receives this depth image, it can directly display the color map if the rendering viewpoint remains unchanged, which leads to the perfect $\mathbf{Q_i}$. If the rendering viewpoint on the client side has been changed, the client can run 3D image warping and display a synthesis immediately [9, 3].

3D image warping is a classic image-based rendering technique firstly proposed by McMillan in [49]. The algorithm takes a depth image $< I^{v_1}, D^{v_1} >$ (including both color map I^{v_1} and depth map D^{v_1}), the viewpoint v_1 at which the depth image is rendered, and the target viewpoint v_2 as input. The output is the color image $W^{v_1 \to v_2}$ at the target viewpoint:

$$W^{v_1 \to v_2} = warping(< I^{v_1}, D^{v_1} >, v_1 \to v_2)$$

The viewpoint is represented by several camera parameters including camera position, view direction, up vector, focal length, and image resolution. For every pixel $I^{v_1}(x_1, y_1)$ in the input image, the algorithm calculates the new coordinate (x_2, y_2) with the following equation:

$$(x_2, y_2) = \left(\frac{a \cdot x_1 + b \cdot y_1 + c + d \cdot \delta_1}{i \cdot x_1 + k \cdot y_1 + k + l \cdot \delta_1}, \frac{e \cdot x_1 + f \cdot y_1 + g + h \cdot \delta_1}{i \cdot x_1 + k \cdot y_1 + k + l \cdot \delta_1} \right) \tag{3.8}$$

where δ_1, also named generalized disparity, is inversely related to the depth value of pixel $I^{v_1}(x_1, y_1)$ and a - l are the variables that can be pre-computed by from the input camera parameters. With the new coordinate, the output image can be generated by copying the color of pixel (x_1, y_1) to the pixel (x_2, y_2) in the target

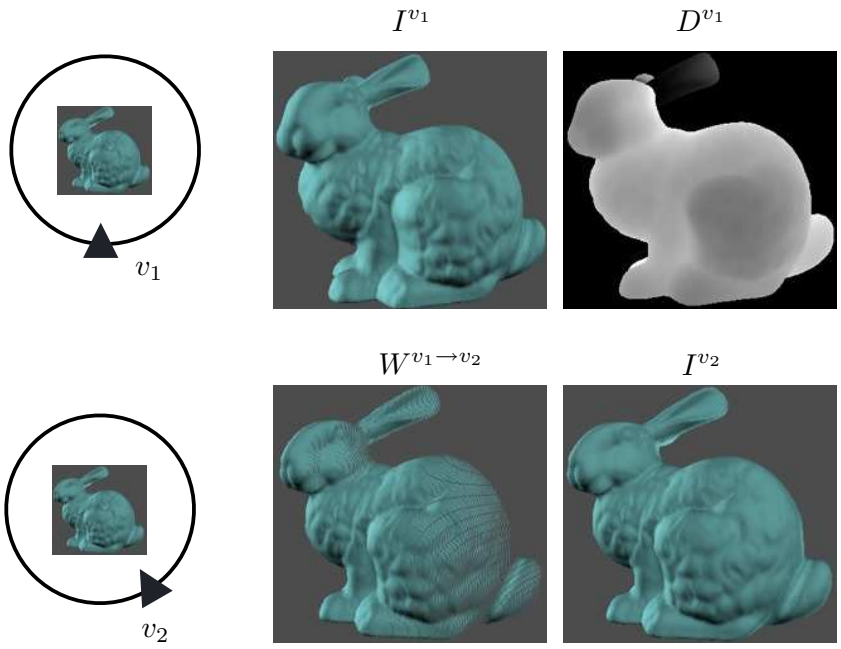


Figure 3.3: Example of 3D image warping

image:

$$W^{v_1 \to v_2}(x_2, y_2) = I^{v_1}(x_1, y_1)$$

Please refer to [48] for more details of the algorithm. Figure 3.3 shows an example of how a bunny is warped to a different viewpoint.

One important feature of 3D image warping is computation efficiency. Calculating the new coordinate requires only 20 arithmetic operations and each pixel of the image is processed only once in this algorithm. Therefore the complexity of the whole algorithm is proportional to the image resolution. The algorithm also handles the situation when multiple pixels in the input image correspond to the same pixel in the output image. McMillan proved that the specific pixel processing order, which is also determined by camera parameters, can guarantee that any pixel that is hidden by other pixel in the output image will always be processed prior to its occluder.

The *Depth Image* approach can be considered as a simplified version of *Point Cloud* because every pixel of the depth image represents a 3D point in the space. The difference is that the depth image only covers the surface of the 3D model at a very limited range, but it also takes much less bandwidth to stream and very light computation to warp.

The *penalty probability* of *Depth Image* is determined by the quality of 3D image warping. We define a

viewpoint set V_{single} as:

$$V_{single} = \{v_x \mid diff(W_i^{vs \rightarrow v_x}, I_i^{v_x}) < err_{resp}\} \tag{3.9}$$

Only if the user interaction changes vc to any viewpoint within V_{single}, the warping result is similar to the rendering result and the interaction latency is reduced (according to Eq. (3.1), $T_{resp} = T(warping)$). So $\mathbf{p} = P(vc \notin V_{single})$.

Multi Depth Image

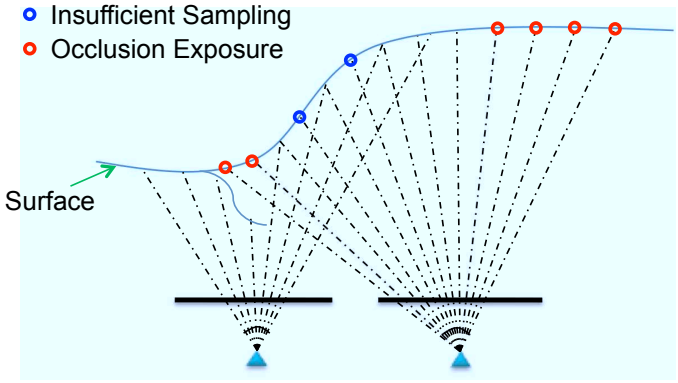
One significant drawback of the *Depth Image* approach is the warping artifacts. The "hole" artifacts are inevitably introduced when one depth image is warped to a different rendering viewpoint because the output image can only use the pixels from the input image. Due to the warping artifacts, V_{single} can not be large enough to cover the possible viewpoints that the user interaction can change to, and the penalty probability $\mathbf{p}$ remains high.

The warping artifacts can be classified into two categories. The first is caused by *occlusion exposure*: the objects that are completely occluded or out of view range in the input image become exposed. The second type of holes is generated due to *insufficient sampling*, which usually happens on a surface with varying gradient. Figure 3.4 shows an example of these two types of warping errors.

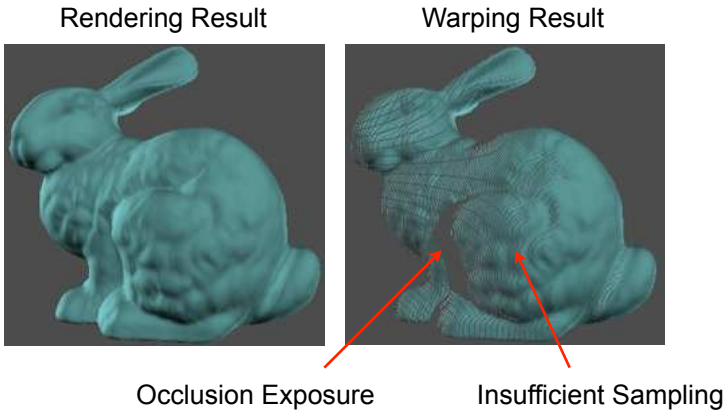
There have been many methods proposed to remove warping artifacts. Depth filter [66] and splats [43] can help fill the small holes caused by insufficient sampling. Super view warping [3] and wide field-of-view warping [43] are useful for out-of-view exposure. Multiple references[43], view compensation [3], and LDI (layered depth image) [75] all use the images from different viewpoints to compensate warping artifacts and can be used to fill both types of holes.

Of all these methods, using multiple references has obvious advantages over others for remote rendering systems. It does not need offline processing like LDI or interactive communication like viewpoint compensation. Figure 3.5 shows an example that two carefully selected references can significantly improve the quality of the warping result. Assume the current server viewpoint is v_0 and the camera orbits right 30 degrees right to v_{30} (the subscript indicates the viewing angle). With only one depth image $< I^{v_0}, D^{v_0} >$, the client can warp it to v_{30} and get the output image shown in Figure 3.5(g) that has a lot of holes. If the server also selects v_{60} as the reference viewpoint and generates one more depth image $< I^{v_{60}}, D^{v_{60}} >$ (Figure 3.5(h)), the client can run 3D image warping twice for two depth images, and composite two low-quality warping results together to get a high-quality synthesis (Figure 3.5(e)) to display.

Using multiple depth images as references can solve the problem of small V_{single}. The server not only generates one depth image at vs, but multiple depth images around vs. By transmitting all the depth



(a)



(b)

Figure 3.4: (a) Illustrate the generation of *occlusion exposure* and *insufficient sampling*; (b) Example of two type of holes in a warping result image

images to the client, the client can either directly display the image if $vs = vc$. Otherwise, the client runs 3D image warping for every depth image and composite the warping result together ([44] proposed a practical composition algorithm). The composition of multiple warping images has much less holes than any single warping result. Similarly to the *penalty probability* analysis before, a viewpoint set V_{multi} is defined as follows:

$$V_{multi} = \left\{ v_x \;\middle|\; diff(\bigcup_k W_i^{ref_k \to vx}, I_i^{v_x}) < err_{resp} \right\} \tag{3.10}$$

$\bigcup_k W_i^{ref_k \to vx}$ is the composition of multiple warping results. The *penalty probability* can be written as $\mathbf{p} = P(vc \notin V_{multi})$. Since V_{multi} is usually much larger than V_{single} (An evaluation is presented in Chapter 4), and therefore more likely to cover the possible viewpoint change. The *penalty probability* of *Multi Depth Image* is much less than *Depth Image*.

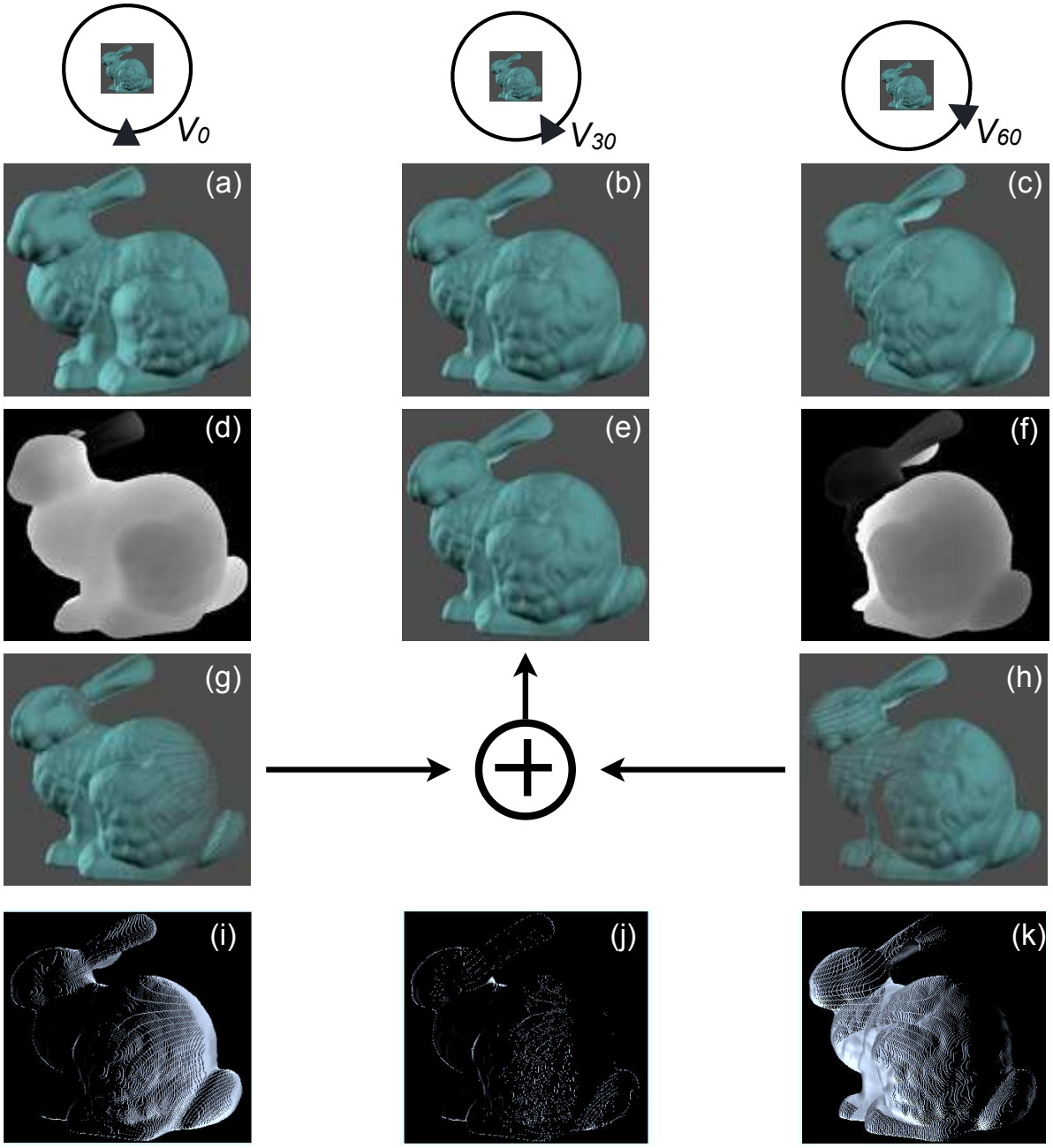


Figure 3.5: The example of using two depth images for 3D image warping: (a) I^{v_0}; (b) $I^{v_{30}}$; (c) $I^{v_{60}}$; (d) D^{v_0}; (e) $W^{v_0 \to v_{30}} \bigcup W^{v_{60} \to v_{30}}$; (f) $D^{v_{60}}$; (g) $W^{v_0 \to v_{30}}$; (h) $W^{v_{60} \to v_{30}}$; (i) $diff(I^{v_{30}}, W^{v_{60} \to v_{30}})$; (j) $diff(I^{v_{30}}, W^{v_0 \to v_{30}} \bigcup W^{v_{60} \to v_{30}})$; (k) $diff(I^{v_{30}}, W^{v_{60} \to v_{30}})$.

Chang's remote rendering system for PDA [9] took a similar approach. The 3D object is pre-rendered at many different rendering viewpoints to generate an LDI [75]. This LDI is streamed to the client for 3D image warping to generate high quality rendering results at a large view range. Besides, the collaborative rendering systems surveyed in Chapter 2 [43, 78, 95] also ran 3D image warping on multiple reference images

Table 3.1: Summary of Remote Rendering Systems (1)

Approaches	**Server Operation: proc_s**	**Client Operation: proc_c**
Original Model	return S_i	$render(S_i, vc)$
Partial Model	generate $S'_i = subset(S_i, vs)$	$render(S'_i, vc)$
Simplified Model	generate $< S'_i, \Delta_i >$, $S'_i = simplify(S_i, vs)$, $\Delta_i = render(S_i, vs) - render(S'_i, vs)$	if $vc = vs$ $render(S'_i, vc) + \Delta_i$ else $render(S'_i, vc)$
Point Cloud	generate $P_i = point\_cloud(S_i, vs)$	$render(P_i, vc)$
Image Impostor	generate $I_i^{vs} = render(S_i, vs)$	return I_i^{vs}
Environment Map	generate $E_i^{vs} = env\_map(S_i, vs)$	if $vc.pos = vs.pos$ $project(E_i^{vs}, vc)$ else do not update display
Depth Image	generate $< I_i^{vs}, D_i^{vs} >$, $I_i^{vs} = render(S_i, vs)$, D_i^{vs} is the depth map of I_i^{vs}	if $vc = vs$ return I_i^{vs} else $W_i^{vs \to vc}$
Multi Depth Image	select a viewpoint set $\{ref_k\}$, let $ref_0 = vs$; for every ref_k, generate $< I_i^{ref_k}, D_i^{ref_k} >$	if $vc = vs$ return I_i^{vs} else $\bigcup_k W_i^{ref_k \to vc}$

Approaches	**Server Proc. Time: T(proc_s)**	**Client Proc. Time: T(proc_c)**
Original Model	0	$T(render)$, depending on S_i complexity
Partial Model	$T(subset)$, depending on the complexity of S_i and the subset selection algorithm	$T(render)$, depending on S'_i complexity
Simplified Model	$T(simplification) + 2 \cdot T(render)$, depending on the complexity of S_i and the model simplification algorithm	$T(render)$, depending on S'_i complexity
Point Cloud	$T(point\_cloud)$, depending on the complexity of S_i and the point cloud generation algorithm	$T(render)$, depending on P_i complexity
Image Impostor	$T(render)$	0
Environment Map	$T(env\_map)$, $\propto T(render)$ (e.g, $T(cubical\_env\_map) = 6 \cdot T(render)$	$T(project)$, $\propto client\_resolution$
Depth Image	$T(render)$	$T(warping)$, $\propto client\_resolution$
Multi Depth Image	$T(ref\_sel) + k \cdot T(render)$, depending on the complexity of viewpoint selection algorithm references and the number of	$k \cdot T(warping)$, $\propto client\_resolution$

to synthesize the high quality results.

The model analysis results of all approaches are summarized in Table 3.1, 3.2, and 3.3. Obviously, of all the approaches analyzed here, *Multi Depth Image* appears to be the best choice to develop a real-time low-latency remote rendering system. Now we present our remote rendering design using this approach.

3.3 System Design

I propose a real-time low-latency remote rendering design that takes the *Multi Depth Image* approach. This new design asks the rendering server to generate more than one depth images as an image-based

Table 3.2: Summary of Remote Rendering Systems (2)

Approaches	**Rendering Quality: Q_i**	**Penalty Probability: p**
Original Model	0	0
Partial Model	depending on how S_i' is created	$P(\mathit{diff}(render(S_i', vc), I_i^{vc}) \geq err_{resp})$
Simplified Model	0	$P(\mathit{diff}(render(\hat{S}_i, vc), I_i^{vc}) \geq err_{resp})$
Point Cloud	depending on how P_i is created	$P(\mathit{diff}(render(P_i, vc), I_i^{vc}) \geq err_{resp})$
Image Impostor	0	1
Environment Map	0	$P(vc.pos \neq vs.pos)$
Depth Image	0	$P(vc \notin V_{single})$, where $V_{single} = \{v_x \mid \mathit{diff}(W_i^{vs \to v_x}, I_i^{v_x}) < err_{resp}\}$
Multi Depth Image	0	$P(vc \notin V_{mul})$, where $V_{multi} = \left\{\forall v_x, \mathit{diff}(\bigcup_k W_i^{vref_k \to v_x}, I_i^{v_x}) < err_{resp}\right\}$

Table 3.3: Remote Rendering Requirements

Approaches	*Latency* Eq. (3.4)	*Quality* Eq. (3.5)	*Real-Time* Eq. (3.6)	*Compression* Eq. (3.7)
Original Model	?	$\surd$	$\times$	$\times$
Partial Model	?	?	?	$\surd$
Simplified Model	$\times$	$\surd$	$\surd$	$\surd$
Point Cloud	?	$\surd$	?	?
Image Impostor	$\times$	$\surd$	$\surd$	$\surd$
Environment Map	?	$\surd$	$\surd$	$\surd$
Depth Image	?	$\surd$	$\surd$	$\surd$
Multi Depth Image	$\surd$	$\surd$	$\surd$	$\surd$

representation of the 3D scene frame in real-time and stream all depth images to the client. The client runs 3D image warping algorithm to synthesize the correct display image.

Although the idea of using multiple references has been proposed and adopted in several projects, including remote rendering systems (e.g. [43, 9]), there are several key differences between my design and and other related systems. First, my remote rendering system generates the depth image references in real-time. Thus it is useful to render not only the static 3D graphics objects, but also the dynamic contents, like 3D video, game scenes, etc. However, the previous system [9] needs offline processing to generate references (LDI) before the remote rendering starts. Second, the collaborative rendering system [43] uses multiple depth images and 3D image warping to reduce the computation workload on the 3D graphics rendering engine and increase the rendering frame rate by interpolating frames in an animation sequence. The goal is maximize the quality of the warping frames given the depth images that are already rendered, so that the

user can not distinguish the warping frames from the rendering frames. However, the goal of my remote rendering system is to reduce interaction latency, which requires the viewpoints of depth image references to be selected even before the user interaction happens. The selection criteria is to maximize the coverage area and maintain adequate warping quality for all the viewpoints in that area. Minor artifacts can be tolerated because the warping result image is only displayed for a short period (rtt) before the server updates arrive.

3.3.1 System Framework

The proposed remote rendering system has a client-server framework. Figure 3.6 and Figure 3.7 illustrate different system modules and how data flows in the system.

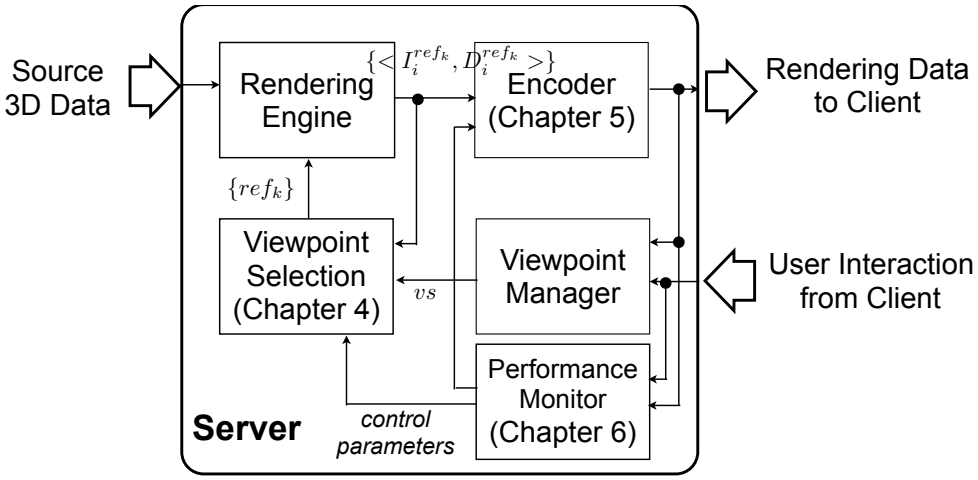


Figure 3.6: The framework of *Multi Depth Image* approach – Server

On the server side, a reference viewpoint selection module selects a viewpoint set $\{ref_k\}$ (let $ref_0 = vs$ all the time for simplicity). The rendering engine needs to render the source 3D data for all viewpoints in $\{ref_k\}$. After rendering, a set of depth image $\{< I_i^{ref_k}, D_i^{ref_k} >\}$ are passed into the encoding module and then transmitted to the mobile client. In addition, there is a performance monitor on the server side to adjust the parameters (e.g., reference selection parameters, encoding parameters, etc) based on the real-time data collected from client and other server modules.

On the client side, if no user interaction happens, or $vc = ref_0$, then $I_i^{ref_0}$ is directly displayed on the screen. Otherwise, the client warps every received depth image to the new viewpoint and composites all the warping results to generate a display image. According to the example in Figure 3.5, the depth images rendered at the carefully selected viewpoints can generate a warping result with much less holes. The user interaction requests are collected on the mobile client and forwarded back to the rendering server in real-

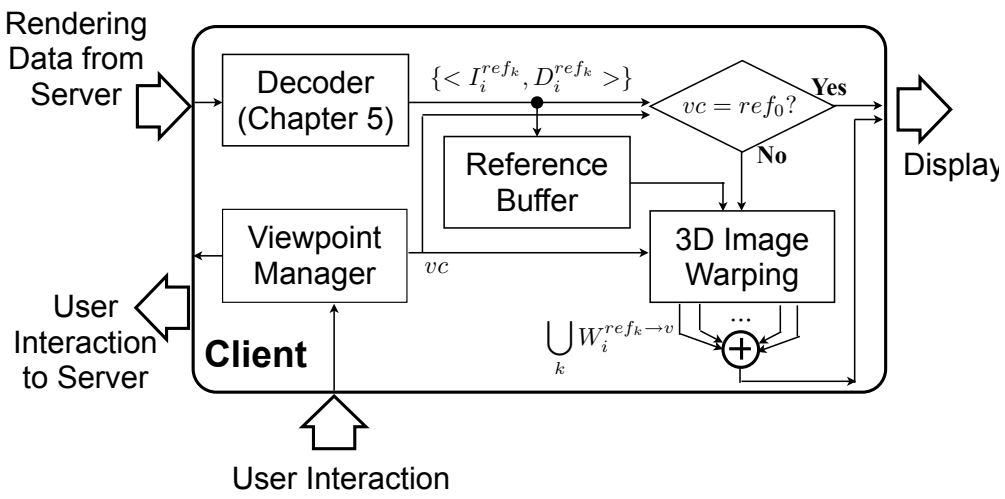


Figure 3.7: The framework of *Multi Depth Image* approach – Client

time. The interaction that changes rendering viewpoints is managed by the viewpoint manager module to produce necessary information (e.g., the camera parameters and variables used in 3D image warping).

3.3.2 Rendering Engine

The rendering engine performs the 3D graphics rendering based on the input 3D source data and the rendering viewpoint fed from the viewpoint selection module. The rendering operations performed in this module are actually determined by the user application. The rendering server provides the underline graphical hardware to accelerate 3D graphics rendering and reuses the engine to generate extra reference frames. Figure 3.10 in the next section explains how to "borrow" the rendering engine from the user application in my implementation.

3.3.3 Viewpoint Manager

Both the server and the client have the viewpoint manager module. The server viewpoint manager receives the user interaction requests output from he client viewpoint manager module and generates the updated rendering viewpoint to the viewpoint selection module. Two types of user interaction requests are supported in the proposed system: viewpoint update and motion update. The viewpoint update contains three vectors: *pos* (camera position), *dir* (look-at direction), and *up* (up direction) to describe the updated rendering viewpoint. For such updates, the viewpoint manger simply forwards the new rendering viewpoint as *vs* to the viewpoint selection module. However, such update is rarely used in our system. The user does not change the rendering viewpoint suddenly to a totally different one unless in some specific scenarios (e.g., the

avatar needs to teleport to a remote place).

Table 3.4: Motion Patterns

Pattern	Description	Direction
Translate	*Pos*: change a unit distance toward $\pm Dir \times Up$ or $\pm Up$; *Dir*: unchanged; *Up*: unchanged.	*Left, Right, Up, Down*
Orbit	*Pos*: change a unit angle in the circle with the tangent toward $\pm Dir \times Up$ or $\pm Up$; *Dir*: remains pointing to the circle center; *Up*: unchanged.	*Left, Right, Up, Down*
Zoom	*Pos*: change a unit distance toward $\pm Dir$; *Dir*: unchanged; *Up*: unchanged.	*Forward, Backward*
Pan & Tilt	*Pos*: unchanged; *Dir*: unchanged; *Up*: change a unit angle in the circle with the tangent toward $\pm Dir \times Up$.	*Left, Right, Up, Down*

The other type of interaction request that contains motion information (e.g., turn right, move forward, stop, etc) is more common. The viewpoint manager needs to interpret such motion updates and calculate the new rendering viewpoint based on application configurations (e.g., moving speed, elapsed time, environment, physical limits, etc). The proposed rendering system currently supports four basic motion patterns illustrated in Table 3.4 and Figure 3.8. These motion patterns are sufficient to cover all basic user interaction requirements in the test applications presented in the next section.

Considering different motion updates may have very different interpretations depending on the application contexts, the viewpoint manager usually reuses the event handlers of the user application to process all motion updates. Refer to Figure 3.10 for implementation details. The client viewpoint manager should process identical operations on the user motion to maintain the consistency between vs and vc.

3.3.4 Viewpoint Selection

The core of the proposed remote rendering design is how to generate reference viewpoints. The viewpoint selection module calculates a set of rendering viewpoints as references based on the current rendering viewpoint from viewpoint manager and the current scene frame (both original 3D data and rendering result are available). Once the reference viewpoints are selected, the depth images rendered at these reference viewpoints will be able to cover a set of viewpoints V_{multi} (Eq. (3.10). As long as vc falls into V_{multi}, the

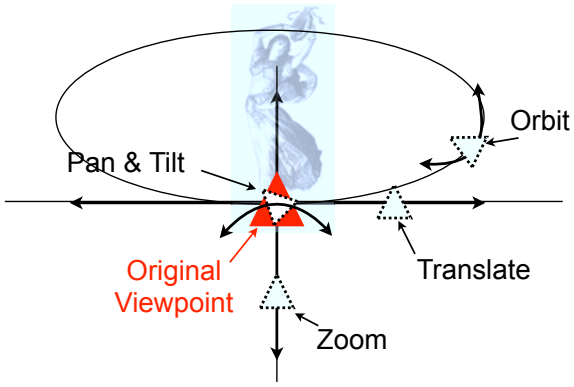


Figure 3.8: User interaction motion pattern

remote rendering system does not suffer from long interaction latency.

However, how user might change the rendering viewpoint is unknown to the viewpoint selection module at the time of selecting reference viewpoints. Therefore, the goal of reference selection is to maximize V_{multi} so that there is a higher probability for vc to fall into the covered area. A reference selection problem can be defined as follows:

> *Given a 3D scene frame S and the current rendering viewpoint vs, find a viewpoint set $\{ref_k\}$ (let $ref_0 = vs$) that maximizes the size of V_{multi} defined in Eq. (3.10.*

I will discuss the solutions to this problem in Chapter 4. Here I introduce some optimization strategies in selecting references for the applications with high refreshing rates in order to avoid generating unnecessary depth images:

1. For the static objects or static scene background, the references are generated only once. The rendering server does not generate new references for the following rendering frames until the viewpoint has been changed. This is because in the static scene, the references can be reused for future frames.

2. When the viewpoint is moving, the rendering server does not generate any references. Instead, the server predicts the position of the rendering viewpoint when the current frame is displayed on the client and renders the frame at the predicted position. In this case, ref_0 does not equal to vs, but vc if the motion has not be changed during the transmission of this frame. In this situation, the client needs to save the history frames in the *reference buffer*. At the moment when the motion is stopped, the client should use the history frames to display or warp before the rendering server receives the new updates.

3. For the dynamic content that every frame is independently created, like 3D video, the references should be generated for every frame unless the subsequent frames are very similar. In this case, the rendering server checks the similarity of every frame by comparing either the source 3D data or rendering result image. The references can be reused only if the differences between subsequent frames are neglectable.

Figure 3.9 shows an example of how the server generates reference frames following the above strategy.

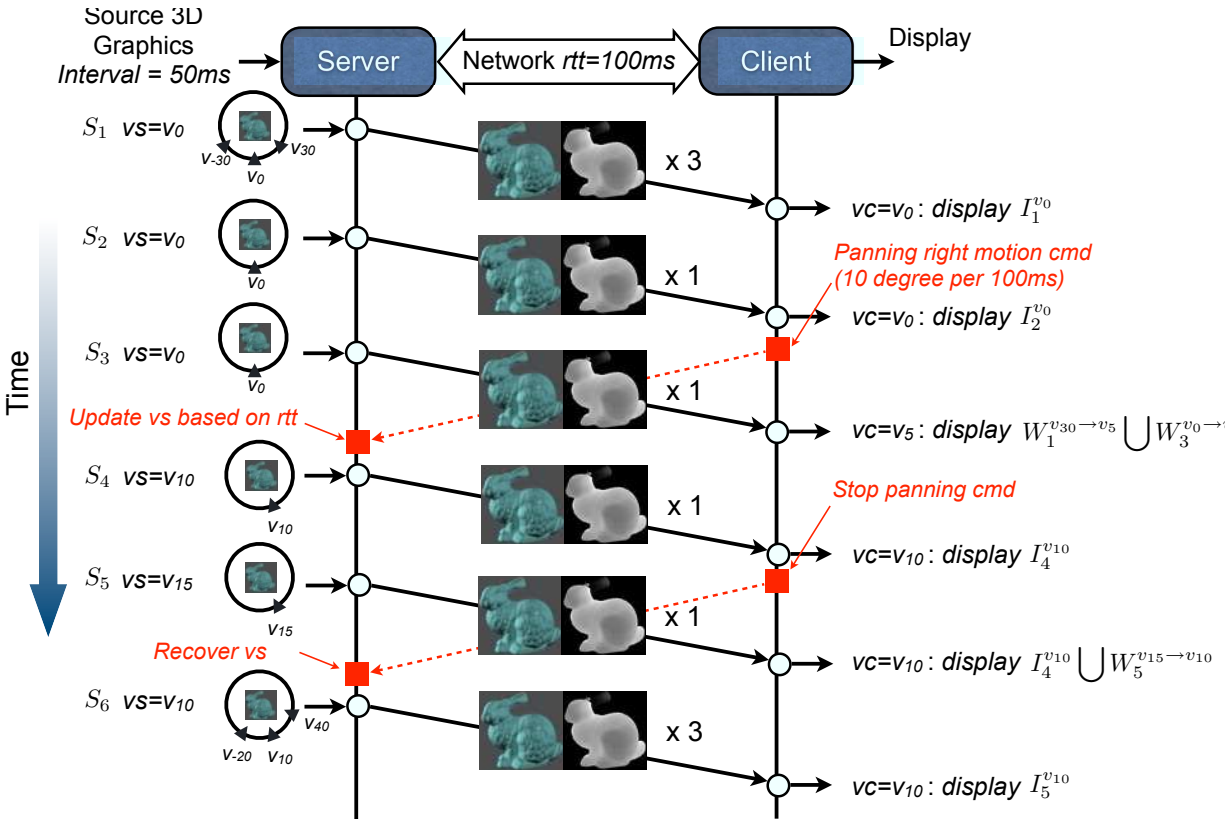


Figure 3.9: An example of reference selection

3.3.5 Encoding/Decoding

The proposed remote rendering system can not perform effectively without an efficient encoder to compress all the extra depth image references. Fortunately, the proposed system generates standard depth images and the compression methods for depth images have been widely studied. The project ATTEST [66] suggests using the standard video codec H.264 to compress depth data. The proxy-based compression proposed by [60] and the skeleton-based compression introduced in [40] both use pre-defined models to reduce data redundancy. Yang [91] proposed two general schemes for intra-stream compression and Kum [32] studied the inter-stream compression using reference stream. These methods can be directly applied in the encoding/decoding module

for depth image compression.

In addition, We have also discovered a new warping assisted coding method that can prominently improve the real-time coding performance in this remote rendering scenario. We will leave the introduction of this novel coding method to Chapter 5.

3.3.6 Performance Monitor

The performance monitor allows us to find out how the rendering system actually performs and tune related parameters of other modules at run-time. We care about not only how much network bandwidth and CPU are being used, but the user experience in playing with our system. One important aspect is the interactive performance: how does the user feel when interacting with the remote rendering system. The details of how to monitor the execution condition and performance, especially the interactive performance of the proposed remote rendering system will be discussed in Chapter 6.

3.3.7 3D Image Warping

This client module runs the standard 3D image warping algorithm for multiple references to generate the final display image. A fixed-point implementation of 3D image warping was proposed in [43]. The hardware warping engine [63] is a good example if the future remote rendering client wants to add a hardware accelerator. The powerful mobile clients can also use GPU to minimize the execution time for 3D image warping [93]. We use the same composition algorithm described in [43] to integrate multiple warping results into one display image and the hole filling algorithms surveyed in [51] to fix small holes with only a few pixels.

3.4 Prototype Implementation

The proposed remote rendering design does not just stay on diagrams, I have implemented a prototype system RRK (Remote Rendering Kit) to study the research problems described in the previous section. A Linux workstation is used as the rendering server and an iPhone 4 smart phone is used as the client. The prototype is not built into the system library so that the graphics application has to be modified for remote rendering. However, the design and implementation of RRK tries to be general purpose so that any standard OpenGL applications can be ported for remote rendering with very little hacking effort. Figure 3.10 illustrates how a standard OpenGL application can interact with RRK to enable remote rendering.

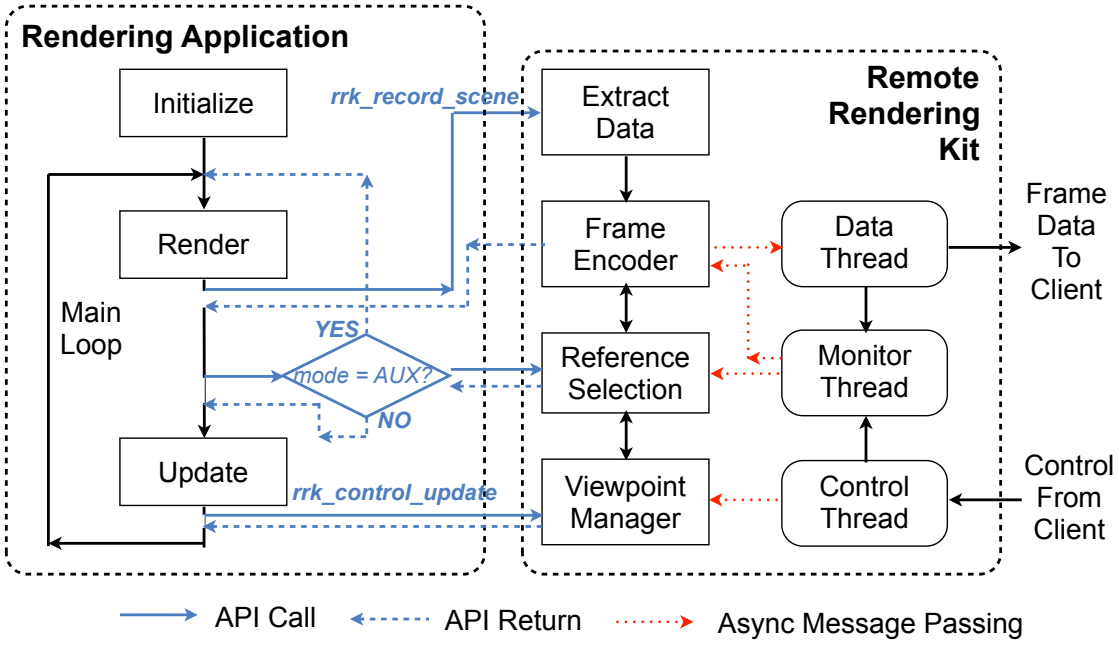


Figure 3.10: Using Remote Rendering Kit

3.4.1 Server Function

The server functions of RRK are packed into a library that provides APIs for the graphics applications to use. Table 3.5 lists the name and usage of several important APIs.

RRK server supports three remote rendering approaches: *Image Impostor*, *Depth Image*, and *Multi Depth Image*. The reference viewpoint set used in the *Multi Depth Image* approach can be selected based on a preset fixed value, or the algorithms discussed in Chapter 4. The image resolution of the reference is set to 480×320 (This resolution is back compatible with earlier iPhone and iPod Touch products). The color or depth images extracted from the rendering engine are downscaled to this resolution before passing to the encoder. Currently, the color images are compressed with JPEG [86] and depth images are compressed with ZLIB [16] (at the time of thesis writing, the coding method proposed in Chapter 5 does not have a real-time implementation). Both TCP and UDP protocols are supported for data transmission between the server and the mobile client.

3.4.2 Mobile Client

On the rendering server, as long as RRK is correctly configured, all server functions run in the background. User only needs to interact with the mobile app. Figure 3.11 shows the user interface I have designed.

In the configuration phase (Figure 3.11(a)), the user can specify the rendering server, streaming protocols,

Table 3.5: Remote Rendering Kit Library APIs

API	Description
`rrk_init`	Initialize the RRK and set up necessary resources (e.g., network ports, memory space, threads, etc) for remote rendering.
`rrk_reg_view_update_cb`	Register a call back function to handle the control signals received from the mobile client.
`rrk_record_scene`	This function must be called every time after the program finishes rendering one frame. Then RRK takes over and extracts the rendering result image from the OpenGL frame buffer. If depth image is needed, the pixel depth values are extracted from *z-buffer*. The camera parameters are calculated from MODELVIEW and PROJECTION matrixes. The frame is then passed to the encoder and sent to the mobile client. If the function is called again before the processing of the previous frame is finished, the function directly returns. Furthermore, the function also manages the rendering viewpoint and rendering mode (see explanations below).
`rrk_rendering_mode`	This function returns the current rendering mode. The mode REF means that the remote rendering system is taking a *Multi Depth Image* approach and the current scene frame should be rendered from other viewpoints. In this case, the main program should bypass all other program logics and execute the frame rendering again. The mode MAIN means that all depth images have already been collected for the current scene frame and the program can continue.
`rrk_control_update`	This function should be called every time before the program starts rendering one frame. RRK uses a separate thread to receive control signals from the mobile client. All received control signals are stored in a queue. When this API is called, the control signals are released from the queue and executed according to the pre-registered call back function to update the rendering viewpoint of the main program.
`rrk_close`	Finalize the RRK library and clean up all resources.

and some control configurations. Once set up, click "connect" and the mobile app will find the server and start remote rendering.

In the rendering phase, the user can interact with applications running remotely on the server. Figure 3.11(b) shows the user interface specially designed for gaming. The bottom left region is the touch area to control the view direction of the avatar (panning/tilting the view camera) and the bottom right region fires the weapon. Three clickable buttons on the top left corner are used to exit, record the game play, and restart the game, respectively. Moreover, the user can move the avatar around by flipping the mobile phone. The accelerometers inside the phone detects the change of the orientation in real-time and translates into control commands.

(a)

(b)

Figure 3.11: Mobile user interface

3.4.3 Applications

Four different graphics applications have been modified to run on the prototype system:

- *Bunny*: The remote visualization of a static 3D graphics model that is constructed by 69,450 triangles (Figure 3.12(a)). The user is allowed to orbit the camera around the bunny and see different parts of the model.
- *Taichi*: This is a 3D video of a tai-chi master giving a class (Figure **??**). recorded by 12 different 3D cameras at the frame rate of 15 fps. Every 3D video frame is reconstructed from 12 depth images. The user can watch the 3D video from different viewpoints.

(a) *Bunny* (b) *Taichi*

(c) *BZFlag* (d) *AlienArena*

Figure 3.12: Remote Rendering Applications

- *BZFlag*: BZFlag[3] is an open source 3D tank battle game (Figure 3.12(c)). The player drives a tank in the battlefield and shoots enemies. The game is played in the first person perspective shooting mode.
- *AlienArena*: AlienArena[4] is another first person shooting game (Figure 3.12(d)). The player control an avatar to move in the game scene and shoot aliens. Compared with *BZFlag*, *AlienArena* renders a more realistic virtual environment and thus requires more complex 3D graphics rendering computation.

The video demo (http://www.youtube.com/watch?v=z-4t9RpdOYg) shows how our prototype system can actually be used by mobile users for real battle fighting. These applications will be used for the evaluation experiments discussed in the following chapters.

3.5 Summary

I have proposed a remote rendering design using the *Multi Depth Image* approach. By comparing different designs head-to-head, *Multi Depth Image* outperforms all other approaches in meeting all the remote rendering requirements specified in this research. However, the idea of using multiple depth image references is still far from building a practically useful real-time remote rendering system that features in high render-

[3]http://www.bzflag.org
[4]http://red.planetarena.org/

ing quality , low interaction latency, and tolerable bandwidth usage. How to select appropriate reference viewpoints and how to efficiently compress the extra depth images are keys to the success of the proposed remote rendering system. In the next two chapters, I will focus on these two issues and present some novel solutions.

Chapter 4

Reference Selection

In this chapter, I will discuss several solutions to the reference selection problem, which is the key to the *Multi Depth Image* remote rendering design. The reference selection can be simplified to a maximization problem. Several search based algorithms and a reference prediction algorithm based on a warping error model are proposed and compared.

4.1 Reference Selection Problem

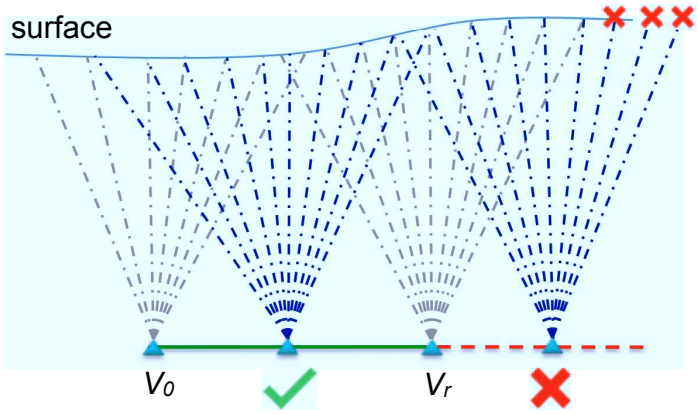
The *Multi Depth Image* approach reduces the interaction latency by providing multiple depth image references for the client to run 3D image warping. However, the problem unsolved is how to select appropriate reference viewpoints. According to the analysis in the previous chapter, the rendering server should find a set of reference viewpoints to maximize V_{multi}:

> *Given a 3D scene frame S and the current rendering viewpoint vs, find a viewpoint set $\{ref_k\}$ (let $ref_0 = vs$) that maximizes the size of V_{multi} defined in Eq. (3.10).*

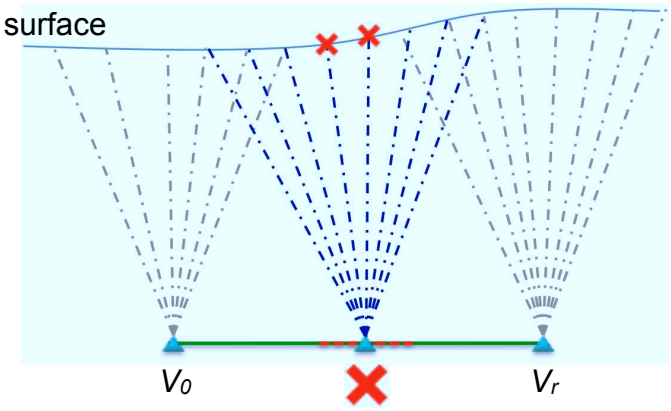
At the time of reference selection, the server does not know what user interaction might happen on the client. Therefore, the server should maximize $\|V_{multi}\|$ to increase the probability that vc will be covered.

Fortunately, the problem can be simplified because the rendering viewpoint in our rendering system does not change freely in the space, but follow specific motion patterns (introduced in Figure 3.8 and Table 3.4). The viewpoint change is always discrete and bounded in distance. We can also assume that the viewpoint motion does not change patterns frequently. These limitations significantly reduce the search space of all possible candidates of vc. For example, given $v_0 = vs$ as the current viewpoint and M is the unit vector of the only motion pattern allowed in the application (the motions of the same pattern but different directions are considered as different motion patterns), the search space of all possible viewpoints can be represented as:

$$V(v_0, M) = \{v_i \mid v_i = v_0 + i \cdot M; \;\; i \in \mathbb{Z} \;\; and \;\; 0 \leq i \leq MAX_M\}$$



(a)



(b)

Figure 4.1: Coverage of two references

where MAX_M is the maximum distance that one user interaction can change the viewpoint.

By selecting one reference viewpoint for every motion pattern, we can convert the original problem of selecting a set of reference viewpoints into the following problem of selecting only one reference:

> *Given a 3D scene frame S, the current rendering viewpoint v_0, and a motion pattern M, find a viewpoint $v_r \in V(v_0, M)$ that maximizes the size of $V_{double}(r)$.*

where $V_{double}(r)$ is defined as:

$$V_{double}(r) = \{v_x \mid v_x \in V(v_0, M) \text{ and } diff(W_i^{v_0 \to v_x} \bigcup W_i^{v_r \to v_x}, I_i^{v_x}) < err_{resp}\} \tag{4.1}$$

With this strategy, the total number of extra depth images in the reference set (the depth image rendered at vs is not counted) equals to the number of motion patterns supported by the rendering system.

In order to solve this simplified problem, let us take an analysis on $V_{double}(r)$ first. Given two references viewpoints v_0 and v_r , the viewpoints $\{v_i \mid 0 \leq i \leq r\}$ take the most benefits from both depth images (illustrated in Figure 4.1(a)). This is because all the pixels of I^{v_i} $(0 \leq i \leq r)$ can be either found in I^{v_0} or I^{v_r}. However, the viewpoints v_j with $j > r$ can not be covered by v_0. The image quality of warping two references is no better than $W^{v_r \to v_j}$. r can not be selected too large either. Otherwise, the viewpoints close to $v_{\frac{r}{2}}$ are too far away from both references and cannot be well covered for high quality warping (illustrated in Figure 4.1(b)). Here we exclude any viewpoints v_j with $j > r$ from $V_{double}(r)$ because the warping results at those viewpoints usually have bad quality (Figure 3.3). Then we have:

$$V_{double}(r) \subseteq [v_0, v_r], \quad [v_0, v_r] = \{v_i \mid v_i \in V(v_0, M), 0 \leq i \leq r\}$$

With the analysis above, the reference selection problem can be further simplified to find the maximum r that still has $V_{double}(r) = [v_0, v_r]$

$$\max_r \{V_{double}(r) = [v_0, v_r]\} \tag{4.2}$$

or described as:

> *Given a 3D scene frame S, the current rendering viewpoint v_0, and a motion pattern M, find the maximum r that maintains $V_{double}(r) = [v_0, v_r]$ when v_r is selected as the reference viewpoint.*

According to Eq. (4.2), the selection of v_r provides a range of viewpoints $[v_0, v_r]$, named as *reference range*. As long as the viewpoint moving distance during an *rtt* (explained in Figure 3.9) is limited within the *reference range*, the *penalty probability* of the remote rendering system is zero. The challenges left are how to find the maximum r and how to find it in real-time.

Note that the $V_{double}(r)$ generated from Eq. 4.2 does not necessarily have the maximum size. In some cases, selecting a even larger r will result in $V_{double}(r) \subset [v_0, v_r]$ but actually a little larger $\|V_{double}(r)|$. However, we can ignore these cases because the pursuit of absolute maximum $V_{double}(r)$ compromises the definition of *reference range*, and makes it more difficult for the rendering server to determine whether the current viewpoint change is covered by the previous references or not.

4.2 Search Based Algorithms

4.2.1 Full Search Algorithm

A baseline *Full Search Algorithm* can be easily derived based on Eq. (4.1) and (4.2). A pseudo code description of the algorithm is in Table 4.1. The algorithm examines every viewpoint v_i $(0 < i \leq MAX_M)$ starting from v_2 as the v_r candidate. For every candidate v_r, all v_x ($0 < x < r$) are verified whether $diff(W^{v_0 \to v_x} \bigcup W^{v_r \to v_x}, I_i^{v_x})$ is less than the threshold err_{resp} (In the algorithm, PSNR is used to measure the difference of two images. $psnr_{resp}$ denotes the PSNR value of threshold err_{resp}. The smaller PSNR value indicates the worse image quality). If every v_x passes the test, which means $V_{double}(r) = [v_0, v_r]$, r is increased and the algorithm continues. If any v_x fails the test, the algorithm stops and returns the previous v_r candidate that successfully passes all the tests. Figure 4.2 shows an illustration.

The *Full Search Algorithm* can find by definition the optimal reference. However, it is obvious that the algorithm can take too much computation to return the optimal result. If the frame rendering and 3D image warping are considered as the unit computation, the *Full Search Algorithm* has the complexity of $O(n^2)$ for 3D image warping and $O(n)$ for frame rendering (Fortunately, the frames rendered for the reference candidates can be reused in the future verification. So the complexity of frame rendering is reduced from $O(n^2)$ to $O(n)$). In the situation that the unit distance of viewpoint change is small, the algorithm takes too long for a real-time system to execute. In our experiments, it actually takes minutes to find the optimal r which is only around 60 for the model *Bunny*.

The algorithm can also be modified for real-time execution by adding a countdown clock. For every execution, the algorithm checks the clock. If the allocated time is used up, the algorithm just stops and returns the latest verified candidate. However, since the server can only verify a few frames in the given short period (e.g., 30 ms), the return value of r is usually too small to be practically useful. Therefore, a more efficient algorithm is needed.

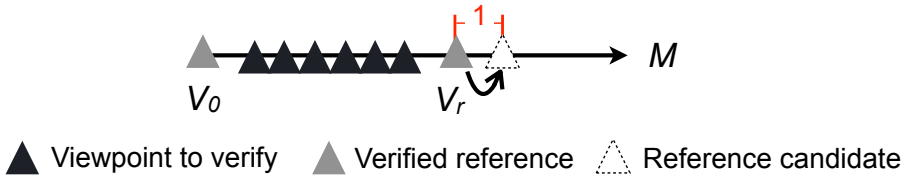


Figure 4.2: Illustration of *Full Search Algorithm*

Table 4.1: Full Search Algorithm

$< I^{v_0}, D^{v_0} >= render(S, v_0)$
$< I^{v_1}, D^{v_1} >= render(S, v_1)$
FOR $r = 2$; $r \leq MAX_M$; $r{+}{+}$
  $< I^{v_r}, D^{v_r} >= render(S, v_r)$
  FOR $i = 1$; $i < r$; $i{+}{+}$
    $W^{v_0 \rightarrow v_i} = warping(< I^{v_0}, D^{v_0} >, v_0 \rightarrow v_i)$
    $W^{v_r \rightarrow v_i} = warping(< I^{v_r}, D^{v_r} >, v_r \rightarrow v_i)$
    $err = psnr(W^{v_0 \rightarrow v_i} \bigcup W^{v_r \rightarrow v_i}, I^{v_i})$
    IF $err < psnr_{resp}$
      RETURN v_{r-1} as the reference viewpoint
    END
  END
END

4.2.2 Median Search Algorithm

The *Full Search Algorithm* checks all intermediate viewpoints for every pair of v_0 and v_r. However, the algorithm only needs one failed verification to stop. Thus, it is only necessary to check the viewpoint with the worst warping quality. The algorithm stops if this "worst" viewpoint fails the test, and otherwise continues. As presented in Figure 4.1(b), the intuitive guess of this "worst" viewpoint is the median viewpoint $v_{\lfloor \frac{r}{2} \rfloor}$.

In order to verify this intuitive assumption, the following experiments have been carried out. I have examined in total $N = 19,908$ reference pairs for 360 different 3D scene frames extracted from *Bunny* application. For every pair of reference viewpoints v_0 and v_r, the maximum warping error err_{max} (the minimum PSNR value) of all intermediate viewpoints is defined as:

$$err_{max} = \min_i \{psnr(W^{v_0 \rightarrow v_i} \bigcup W^{v_r \rightarrow v_i}, I^{v_i}) \mid k \in (0, r)\}$$

A score P_i is given for each intermediate viewpoint:

$$P_i = \begin{cases} 1 & if \;\; psnr(W^{v_0 \rightarrow v_i} \bigcup W^{v_r \rightarrow v_i}, I^{v_i}) = err_{max} \\ 0 & else \end{cases}$$

Of all N tests, the probability of each intermediate viewpoint v_i to have the maximum warping error can be described with a random variable $p_{max}(i)$:

$$p_{max}(i) = \frac{\sum P_i}{N}$$

The intermediate viewpoints of all tests are normalized by setting the median viewpoint $v_{median} = v_{\lfloor \frac{r}{2} \rfloor}$

to the origin point. Figure 4.3 shows the PDF function of $p_{max}(i)$. Obviously, v_{median} has the highest probability to generate maximum warping error of all intermediate viewpoints.

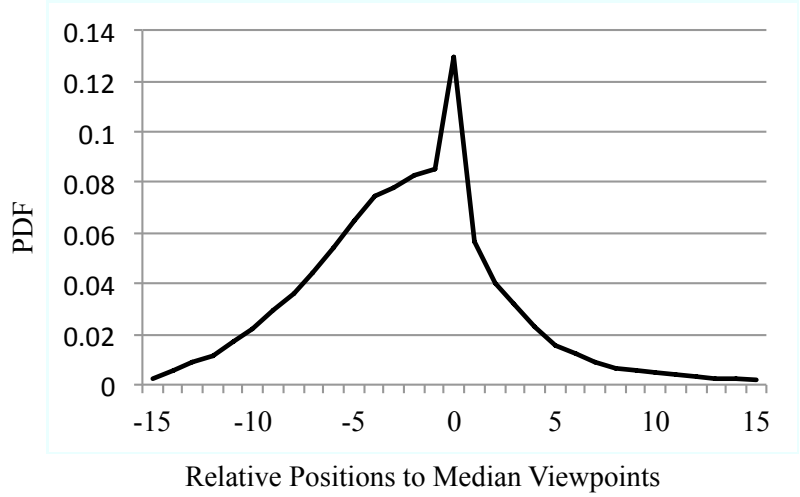


Figure 4.3: Probability of Maximum Warping Error

The *Median Search Algorithm* is described in Table 4.2. For each pair of viewpoints, we only calculate the warping error of v_{median}. Figure 4.4 shows an illustration. The algorithm successfully reduces the complexity of 3D image warping to $O(n)$. Although there exist exceptions that median search will result in larger *reference range* than the optimal result, the error is considered tolerable. It is because the algorithm still has a complexity of $O(n)$ for frame rendering and it is unlikely to find a very large r in the scenario of real-time rendering.

Table 4.2: Median Search Algorithm

$< I^{v_0}, D^{v_0} >= render(S, v_0)$
$< I^{v_1}, D^{v_1} >= render(S, v_1)$
FOR $r = 2$; $r \leq MAX_M$; r++
  $< I^{v_r}, D^{v_r} >= render(S, v_r)$
  $median = \lfloor \frac{r}{2} \rfloor$
  $W^{v_0 \rightarrow v_{median}} = warping(< I^{v_0}, D^{v_0} >, v_0 \rightarrow v_{median})$
  $W^{v_r \rightarrow v_{median}} = warping(< I^{v_r}, D^{v_r} >, v_r \rightarrow v_{median})$
  $err = psnr(W^{v_0 \rightarrow v_{median}} \bigcup W^{v_r \rightarrow v_{median}}, I^{v_{median}})$
  IF $err < psnr_{resp}$
    RETURN v_{r-1} as the reference viewpoint
  END
END

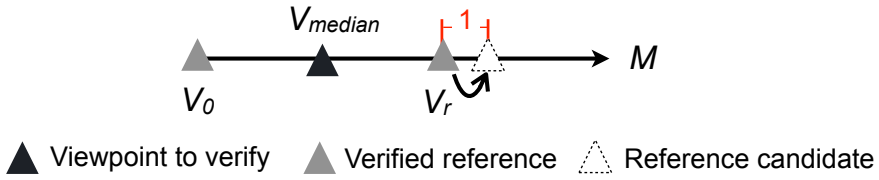


Figure 4.4: Illustration of *Median Search Algorithm*

4.2.3 Fast Median Search Algorithm

The *Median Search Algorithm* can be optimized further. When the median viewpoint meets the requirement, the current algorithm simply increases the r by one and continues the next test. The algorithm can be significantly speeded up by increasing the searching step exponentially rather than one step a time. Moreover, the initial value of the reference viewpoint r can be set to a large number rather than 2. It avoids the computation to verify small *reference ranges* because they usually pass the test easily.

Table 4.3 has the description of the *Fast Median Search Algorithm*. It combines all the optimizations discussed above. The algorithm is divided into two phases. At the beginning, the search stride increases exponentially until the any candidate fails the test. In the second phase, the stride decreases exponentially to find a accurate position of where the reference viewpoint should be. Figure 4.5 shows an illustration. The *init_stride* depends on the motion unit distance configured in the rendering system (In my system, it is set to 8). The algorithm complexity has been reduced to only $O(\log(n))$ for both frame rendering and 3D image warping.

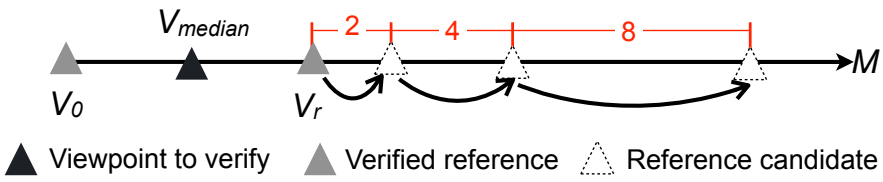


Figure 4.5: Illustration of *Fast Median Search Algorithm*

4.2.4 GPU Implementation

Another important feature of the proposed algorithm is that it can be easily decomposed into individual elements and executed in parallel. Here I briefly discuss how to implement these search based algorithms using CUDA.

CUDA is a parallel computing architecture developed by NVIDIA. It manages the GPU computation in a

Table 4.3: Fast Median Search Algorithm

$< I^{v_0}, D^{v_0} >= render(S, v_0)$
$median = \lfloor \frac{init\_stride}{2} \rfloor$
$< I^{v_{median}}, D^{v_{median}} >= render(S, v_{median})$
$stride = init\_stride$
$r' = 0$
WHILE TRUE
  $r = stride$
  $< I^{v_r}, D^{v_r} >= render(S, v_r)$
  $median = \lfloor \frac{r}{2} \rfloor$
  $W^{v_0 \rightarrow v_{median}} = warping(< I^{v_0}, D^{v_0} >, v_0 \rightarrow v_{median})$
  $W^{v_r \rightarrow v_{median}} = warping(< I^{v_r}, D^{v_r} >, v_r \rightarrow v_{median})$
  $err = psnr(W^{v_0 \rightarrow v_{median}} \bigcup W^{v_r \rightarrow v_{median}}, I^{v_{median}})$
  IF $err \geq psnr_{resp}$
    $r' = r$
    $stride = 2 \times stride$
  ELSE
    $stride = \frac{stride}{4}$, BREAK
  END
END
WHILE $stride > 0$
  $r = r' + stride$
  $< I^{v_r}, D^{v_r} >= render(S, v_r)$
  $median = \lfloor \frac{r}{2} \rfloor$
  $W^{v_0 \rightarrow v_{median}} = warping(< I^{v_0}, D^{v_0} >, v_0 \rightarrow v_{median})$
  $W^{v_r \rightarrow v_{median}} = warping(< I^{v_r}, D^{v_r} >, v_r \rightarrow v_{median})$
  $< I^{v_{median}}, D^{v_{median}} >= render(S, v_{median})$
  $err = psnr(W^{v_0 \rightarrow v_{median}} \bigcup W^{v_r \rightarrow v_{median}}, I^{v_{median}})$
  IF $err \geq psnr_{resp}$
    $r' = r$
    $stride = \frac{stride}{2}$
  ELSE
    $stride = \frac{stride}{2}$
  END
END
RETURN $v_{r'}$ as the reference viewpoint

way that provides a simple interface for the programmer. In the implementation of the proposed algorithms, the dimension of CUDA kernels (e.g., the number of thread blocks, grids, etc.) is determined based on the resolution of reference frames. Different memory models are utilized to efficiently manage data in processing. Typically, since the size of color and depth map rendered for 3D image warping is larger than the shared memory (typically 16KB in a single thread block), they are stored in either shared or global memory which can be accessed by multiple threads in multiple grids. From the 3D image warping equation (Eq. (3.8)), each thread with partitioned pixels and shared depth information calculates a part of new coordinates (i.e., indices in matrices) of the warped image frame, and the full frame based on the new coordinate is merged together for further use.

Compared with executing reference search algorithms on CPU, GPU implementation has two important benefits. The first is the scalability as the frame sizes getting larger, hundreds of GPU cores provide far more computation resources than CPU to exploit the maximum computational parallelism on the given data. The second advantage is data transfer. Since the input frames for 3D image warping are exported from the frame buffer and *z-buffer* of the graphics pipeline, the GPU implementation saves the large data transfer between CPU memory and GPU memory. Refer to [93] for more implementation details and experiment data.

Note that, although GPU provides a parallel environment that greatly accelerates the algorithm execution, it is not able to guarantee the real-time performance as the 3D rendering frame rate increases or the 3D source content becomes more complicated. In the scenario that strong real-time requirements are enforced, the algorithm has to stop and return the current reference candidate as the reference viewpoint when the allocated time slot is used up.

4.3 Reference Prediction Algorithm

In this section, I will introduce a new algorithm to select reference viewpoint. Instead of searching all possible candidates, the novel algorithm predicts the position of the position of the targeted reference viewpoint based on a warping error model. Therefore, it does not need to render any extra frames or run 3D image warping. Compared with the previous search based algorithms, this new *Reference Prediction Algorithm* requires much less computing resources from the rendering server and is a perfect match to our real-time remote rendering system.

4.3.1 Warping Error Function

The algorithm is derived from an error function $F(r)$ that predicts the maximum warping error for any viewpoint in the range $[v_0, v_r]$ when v_r is selected as the reference. The function is defined as follows:

$$F(r) = \sum_{w=0}^{W-2} \sum_{h=0}^{H-1} f(w,h,r), \quad 1 < r \leq MAX_M \tag{4.3}$$

where W and H are the width and height of the generated image. $f(w,h,r)$ is the pixel pair error function for the adjacent pixels (w,h) and $(\hat{w},\hat{h})$[1] in I^{v_0}.

$$f(w,h,r) = \begin{cases} 0 & if \quad h_{max}(w,h) < 1 \quad or \quad r \leq i_{max}(w,h) \\ \lfloor h_{max}(w,h) \rfloor \cdot \left(I^{v_0}(\hat{w},\hat{h}) - I^{v_0}(w,h) \right)^2 & else \end{cases} \tag{4.4}$$

$I^{v_0}(w,h)$ returns the intensity of pixel (w,h). h_{max} and i_{max} are explained as follows.

The warping error can be estimated on the base of pixels. The holes are introduced because the pixels that are adjacent in the original picture are not adjacent any more after 3D image warping. For every pair of adjacent pixels (w,h) and $(\hat{w},\hat{h})$, we can build a hole size function $h(i)$, $2 \leq i \leq MAX_M$ to represent the size of the hole caused by warping this pair of pixels to viewpoint v_i. For example, Figure 4.6 shows a scenario that the viewpoint translates from v_0 right to v_i horizontally. A and B are two points on the surface visible at both v_0 and v_i. d_a and d_b are the depth value of A and B to the image plane. a_0 and b_0 are the horizontal coordinates[2] of the corresponding pixels of A and B in I^{v_0}. If $b_0 = a_0 + 1$, then A and B are adjacent in I^{v_0}. The horizontal coordinates a_i and b_i in I^{v_i} can be represented as:

$$a_i = a_0 - \frac{i \cdot M \cdot f}{d_a}, b_i = b_0 - \frac{i \cdot M \cdot f}{d_b}$$

where M is the unit vector of viewpoint translation and f is the distance from center-of-projection to the image plane. The hole size function $h(i)$ in this case is:

$$\begin{aligned} h(i) &= (b_i - a_i) - (b_0 - a_0) \\ &= M \cdot f \cdot i \left(\frac{1}{d_a} - \frac{1}{d_b} \right) \end{aligned} \tag{4.5}$$

Define h_{max} as the max value that $h(i)$ can reach within the given viewpoint range ($i \in [2, MAX_M]$) and i_{max} as the viewpoint position at which h_{max} is achieved. In the example scenario (Figure 4.6), if $d_a < d_b$, the hole size increases monotonically with i and reaches the max value when $a_i = 0$. Thus:

$$i_{max} = \frac{d_a \cdot a_0}{M \cdot f} \tag{4.6}$$

$$h_{max} = a_0 \left(1 - \frac{d_a}{d_b} \right) \tag{4.7}$$

[1] $(\hat{w},\hat{h})$ denotes the adjacent pixel to (w,h) and for different motion patterns it has different definitions. For example, for horizontal movements, $\hat{w} = w + 1$, $\hat{h} = h$; and for vertical movements, $\hat{w} = w$, $\hat{h} = h + 1$. For the pixels on the image edge which do not have valid $(\hat{w},\hat{h})$, $f(w,h,r)$ always returns 0.

[2] the vertical coordinates are neglected here because there is no viewpoint movement in the vertical direction

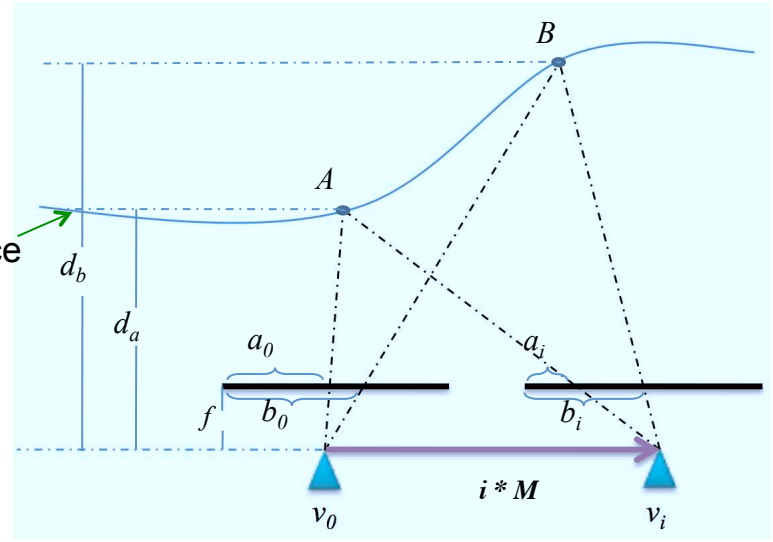


Figure 4.6: Warping Hole Size Analysis

Back to the definition of pixel pair error function in Eq. (4.4). For a given pixel pair, $h_{max} > 1$ means this pixel pair may create a visible hole after 3D image warping. Therefore, given the current viewpoint v_0 as one reference, the other reference v_r should provide the pixels missing from v_0. Intuitively, the best viewpoint that can provide the most missing pixels is where the warping result has the largest holes. In order to fill this hole, the reference image at v_r should have all the missing pixels so that i_{max} is the best candidate for r. If $r > i_{max}$, the reference viewpoint v_r may not have enough pixels to cover the warping hole at $v_{i_{max}}$ and an error score is contributed by this pixel pair to the overall error function. The error score (equivalent to Mean Square Error) is aggressively calculated based on the pixel difference and hole size. This is because a hole filling technique is usually applied to fill the holes with its neighbor pixels. In the worst case, the hole can be completely filled with the wrong neighbor pixel.

4.3.2 Reference Prediction

With the error function, I propose the *Reference Prediction Algorithm*. Table 4.4 describes the algorithm with pseudo codes. First, h_{max} and i_{max} are calculated for every pair of adjacent pixels in I^{v_0}. Then the error function F is tested for every reference candidate. If the total error score is larger than the pre-set threshold, the algorithm stops and return the previous candidate as the predicted reference. Compared with the search based algorithms, the *Reference Prediction Algorithm* does not generate the optimal result but requires much less computation: only I^{v_0} is rendered, no 3D image warping is performed, and the algorithm

complexity is equivalent to scanning the whole image.

Table 4.4: Reference Prediction Algorithm

$< I^{v_0}, D^{v_0} >= render(S, v_0)$
FOR $w = 0$; $w < W$; w++
  FOR $h = 0$; $h < H$; h++
    calculate $h_{max}(w,h)$, $i_{max}(w,h)$ for I^{v_0}
    IF $h_{max}(w,h) < 1$
      CONTINUE
    END
    $F[i_{max}(w,h)+1] = F[i_{max}(w,h)+1] + h_{max}(w,h) \times (I^{v_0}(w,h) - I^{v_0}(adj_w(w), adj_h(h)))^2$
  END
END
FOR $r = 2$; $r \leq MAX_M$; r++
  $F[r] = F[r] + F[r-1]$
  IF $F[r] > mse_{disp}$
    RETURN v_{r-1} as the reference viewpoint
  END
END

4.4 Evaluation

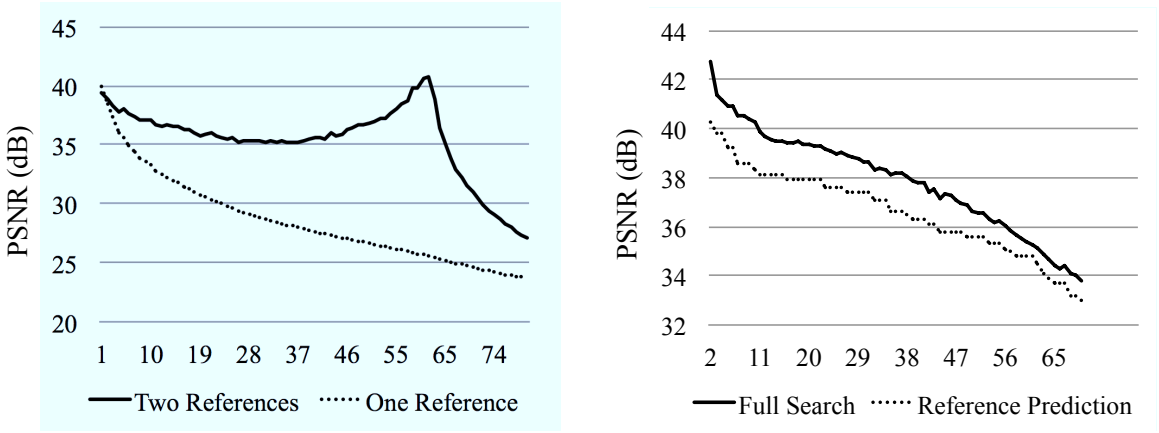
The experiments to evaluate the performance of three search based algorithms use a workstation with an Intel Quad-Core Xeon processor running at 2.33GHz, 3GB memory, and an NVIDIA GeForce 9800 GX2 GPU with 256 cores and 1GB of memory. The algorithms are implemented with CUDA SDK 2.1. The 3D video stream *Taichi* is used as the test application. Considering *Taichi* should be rendered at 15 fps, there is a strict real-time requirement to finish all reference selection, 3D graphics rendering, and image frame compression within the frame time, 66ms in this case. Table 4.5 compares the average *reference range* of all three algorithms. Since none of the algorithm can finish before the deadline, a timer is added to the implementation to force the algorithm to quit and return the current reference candidate when time expires. Obviously, with the same computation resources, the *Fast Median Search Algorithm* generates the largest *reference range*.

Table 4.5: Performance of Search Based Algorithms

Display Resolution	Frame Rate(fps)	Reference Range		
		Full	Median	Fast
176 × 144	15	5	6	15
320 × 240	15	3	4	14
640 × 480	15	1	2	12

Figure 4.7(a) plots the image quality of warping results. The horizontal axis indicates the viewpoint position and the vertical axis is the PSNR of the warping image at the given viewpoint. For the line of *One*

Reference, only v_0 is used as the reference viewpoint. The PSNR drops fast as the target viewpoint moves away from the reference. For the line of *Two references*, v_0 and v_{60} are used as references. High warping quality is maintained for all viewpoints between two references. If we set 35 dB as the threshold $psnr_{resp}$, the size of V_{single} is only 3 while the size of V_{double} is 63.



(a) (b)

Figure 4.7: (a) The warping quality comparison between one depth image as reference and two depth images as references; (b) The comparison of reference viewpoint selection algorithms.

Figure 4.7(b) presents how reference viewpoint is predicted. The horizontal axis stands for the reference candidates. For the line of *Full Search*, the vertical axis indicates the PSNR of the worst warping image inside the reference range:

$$PSNR(x) = \min\left\{\forall v_i \in [v_0, v_x], psnr(W^{v_0 \to v_i} \bigcup W^{v_x \to v_i}, I^{v_i})\right\}$$

For the line of *Reference Prediction*, the vertical axis is the PSNR equivalent score of $F(x)$ as defined in Eq. (4.3). If we set 35 dB as the threshold of $psnr_{resp}$, we get 62 as the output of *Full Search Algorithm* and 54 as the output of *Reference Prediction Algorithm*.

Although *Reference Prediction Algorithm* does not generate an optimal result, it is much faster in execution. In our experiments, it takes a couple of minutes to finish the whole *Full Search Algorithm* and generate the optimal result but only a few milli-seconds to run *Reference Prediction*.

4.5 Summary

The idea of using multiple depth images as references in a remote rendering system to reduce interaction latency depends on how the reference viewpoints are selected. In this chapter, I have proposed several

algorithms to address a simplified version of this reference selection problem: select one reference viewpoint for a given motion pattern to maximize the *reference range*. Depending on how many motion patterns the rendering system supports, the reference selection algorithm should be executed multiple times to generate multiple reference viewpoints within the rendering time of a single frame. Therefore, only the last *Reference Prediction Algorithm* can actually be applied in a real-time remote rendering system.

Chapter 5

Data Compression

A 3D image warping assisted real-time video coding method is introduced in this chapter. The proposed coding method takes advantage of the graphics rendering contexts (depth maps, camera parameters, and camera motion) to improve the real-time video coding performance. This novel method fits best for the cloud gaming, and can also serve for other remote rendering applications, like remote visualization and 3D video rendering.

5.1 Introduction

A good remote rendering system does not compress every rendering frame individually, but encode frames as video. The modern video coding standards are all based on hybrid video coders, which exploit not only spatial correlation using transform coding, but also temporal redundancy using motion prediction [87, Section 9.3]. In order to meet the real-time requirement of remote rendering, only the real-time video coder can be applied in the remote rendering system.

Unlike general purpose offline video coders that employ long look-ahead coding windows and complex coding tools for high coding efficiency, real-time video coders, such as [84, 5, 41], work under several constraints [65, 74]:

- There is no bidirectional prediction because B frames need to be saved in the buffer until the future P frame shows up;
- The two-pass rate control is inherently infeasible for the remote rendering scenario. Thus only the one-pass rate control is used;
- The coder should configure low buffer occupancy and intra refresh coding to ensure that individually coded frames do not exceed a maximum size, and thus coded frames can be directly streamed to the network without being buffered;

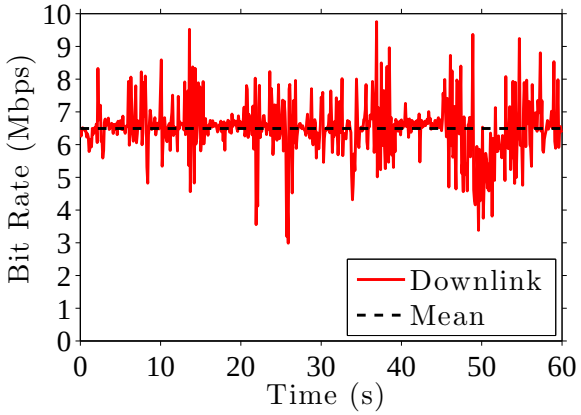
- The encoding deadline is strict. According to [37, 52], the extreme low coding delay can be achieved at an expense of significantly lower coding efficiency.

Because of the aforementioned constraints, compared to general purpose video coders, real-time video coders suffer from a much lower coding efficiency [24]. To illustrate the coding inefficiency of real-time video coders, we have conducted the following experiment on OnLive [57], the state-of-art cloud gaming service provider. According to [61], the game rendering server of OnLive uses a customized H.264 real-time coder to encode all game scenes. We installed the OnLive thin client on a dual-core laptop, which was connected to 10Mbps Ethernet. A first-person shooting game *Unreal Tournament III: Titan Pack* (Figure 5.1(a)) was played for 60 seconds, and all downlink network packets were captured. We also captured the 720p video from the laptop's frame buffer, and saved it as a raw video file. From the network trace, we plot the OnLive streaming rate in Figure 5.1(b), which indicates that real-time OnLive encoder produces a stream at a fairly high bit rate: 6.49Mbps on average. Next, we encoded the raw video at several bit rates using x264 [24], which is a general purpose H.264/AVC encoder. We plot the resulting rate-distortion curve in Figure 5.1(c). This figure shows that x264 can encode the OnLive stream very well at rather low bit rates: x264 achieves 40+ dB at 1Mbps. In summary, Figure 5.1 reveals that real-time video coding is very challenging; even the state-of-art OnLive video coder suffers from coding inefficiency.

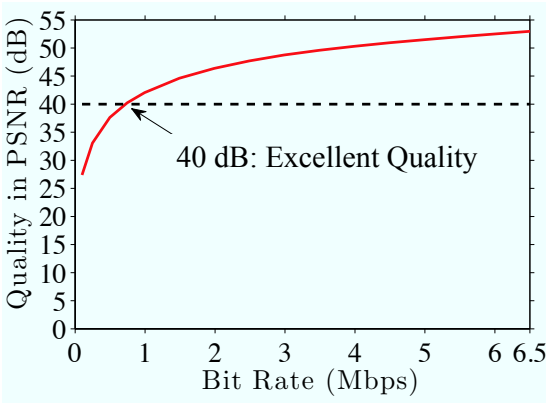
Rather than focusing only on the video coding component, I took a different look at the problem from a system's perspective and found a new approach. I propose a 3D image warping assisted real-time video coding method. The method takes advantage of the graphics rendering contexts (rendering viewpoint, pixel depth, and camera motion) because the video encoder runs together with the graphics engine on the rendering server. The basic idea is to select key frames from the frame sequence, and then use the 3D image warping algorithm to interpolate other intermediate frames. Finally, H.264/AVC is used to encode all key frames and warping residues (the difference between the interpolation result and the original image frame) for each intermediate frame. The interpolation allows us to encode warping residues with a much lower bit rate, and assign more bits to encode key frames. In the ideal situation, if all intermediate frames are perfectly interpolated and no residues need to be encoded, all bit rates can be assigned to encode key frames for better video quality. Thus, the encoding performance of the proposed method depends on the quality of the interpolation method.

As I have introduced in previous chapters, 3D image warping is a well-known image-based rendering (IBR) technique and fits in our coding method very well. Given the pixel depth and rendering viewpoint, the algorithm can efficiently warp an image to any new viewpoint. The depth and viewpoint information are the graphics contexts accessible from the rendering engine. It is similar to the motion estimation

(a)



(b)



(c)

Figure 5.1: (a) Screenshot of *Unreal Tournament III*, (b) OnLive streaming rate, and (c) coding efficiency of a general purpose video coder.

method widely used in video coding tools. The difference is that motion estimation requires intensive search computation between the two image frames and the motion vector has a limited range in the scenario of low delay real-time coding. On the contrary, 3D image warping is efficient in computation and capable of adapting to any wild motion. More importantly, the proposed coding method can be used to compress the depth image references seamlessly. In the following sections, I will present how the depth image references can be directly used as key frames and how the reference selection algorithms discussed in Chapter 4 can be useful in selecting key frames.

5.2 Related Work

Giesen et al. [25] took a very similar approach to mine. They tried to improve the performance of x264 encoding by building in 3D image warping to assist motion estimation. Their approach takes advantage of x264 to efficiently compress all warping residues, but also suffers from processing 3D image warping in blocks. As a comparison, my method warps the image at pixel level and our major concern is to select

appropriate reference frames based on camera motion.

Mark [43] added a 3D image warping module to the conventional graphics pipeline to increase the rendering frame rate. Only key frames are rendered and other intermediate frames are interpolated by 3D image warping. Although it is an opposite problem to mine, which is to "remove" the actually rendered frames, many of my approaches, such as using multiple reference frames and how to generate reference frames, are inspired by Mark's work. The core idea behind both works is to exploit the frame-to-frame coherence in graphics rendering with image based rendering techniques.

Some researches in the area of 3D/multi-view video coding [50, 76] are also related in using the same tool to build coders. However, there are some major differences between my work and 3D/multi-view video coding . First, 3D/multi-view video coding approaches only use IBR techniques to deal with different views at the same time spot while I apply 3D image warping to synthesize frames of different time. Second, in order to obtain accurate camera parameters for IBR, the cameras used in 3D/multi-view coding are usually statically deployed. Thus, only the object movements are studied and no camera motion needs to be considered. Besides, the depth capturing is usually not accurate enough for high quality IBR. On the contrary, my coder is able to extract accurate camera and depth information directly from 3D graphics rendering engine. Last, in 3D/multi-view video coding, only the actually captured frames can be selected as reference frames. But my coder can select the frames that do not exist in the video sequence as the reference frame.

My work is also different from the projects that apply 3D image warping techniques to generate stereo views from 2D sources [66]. It is because the 2D-3D conversion systems usually focus on high quality warping while my coder concentrates on coding performance improvement. This key difference leads to very different frame selection strategies and system framework.

5.3 3D Image Warping Assisted Video Coding

In this section, I present how the proposed 3D image warping assisted video coding method works.

5.3.1 Overview

To give one sentence overview of the proposed coding method: "Select a set of key frames (named R frame) in the video sequence based on the graphics rendering contexts extracted from the 3D video game engine, use the 3D image warping algorithm to interpolate other intermediate frames (named W frame) with the selected R frames, and encode the R frames and warping residues of W frames with x264". The proposed method can improve the coding performance by assigning more bit rate to encode the more important R

frames and less bits for W frame residues. Figure 5.2 and 5.3 show the framework in block diagrams and how data flows.

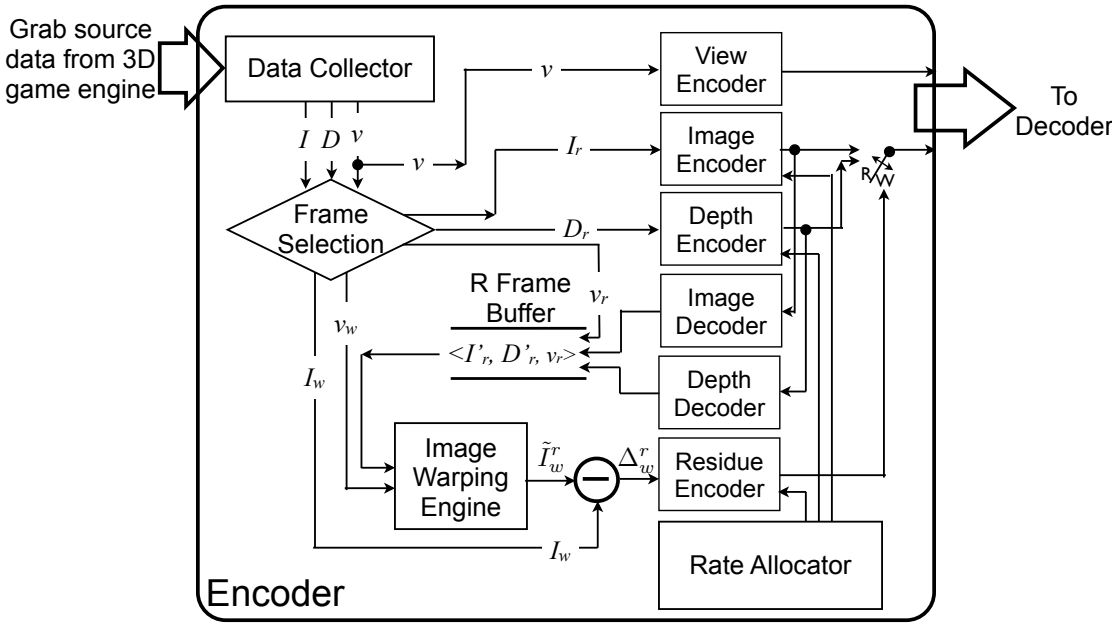


Figure 5.2: Framework of the 3D image warping assisted video encoder

The core idea of the method is to exploit the frame-to-frame coherence. It is similar to motion estimation in many ways. For example, the concept of R/W frame is close to I/P frame in motion estimation. However, with the support of graphics rendering contexts, our method runs much faster than the search based motion estimation algorithms, and thus is more efficient in the real-time remote rendering scenario.

In the following subsections, I will introduce different components in the proposed coding method, starting with 3D image warping.

5.3.2 3D Image Warping

The 3D image warping algorithm has been reviewed in Section 3.2.2. Here I explain how to apply the 3D image warping algorithm to assist video coding.

Given a source video frame set $\{I_x|x \in \mathcal{S}\}$, if we also know the depth map $\{D_x|x \in \mathcal{S}\}$ and viewpoint $\{v_x|x \in \mathcal{S}\}$ of each frame[1], we can select a group of R frames as $\mathcal{R}$ and the rest frames are all W frames as $\mathcal{W}$. The warping version $\{W_x^{v'_{ref(x)} \rightarrow v_x}|x \in \mathcal{W}\}$ can be generated by running 3D image warping algorithm

[1]Note that definition of $\{v_x\}$ is different from the previous chapter. v_i is the rendering viewpoint at which $< I_x, D_x >$ (simple for $< I_x^{v_x}, D_x^{v_x} >$) are rendered.

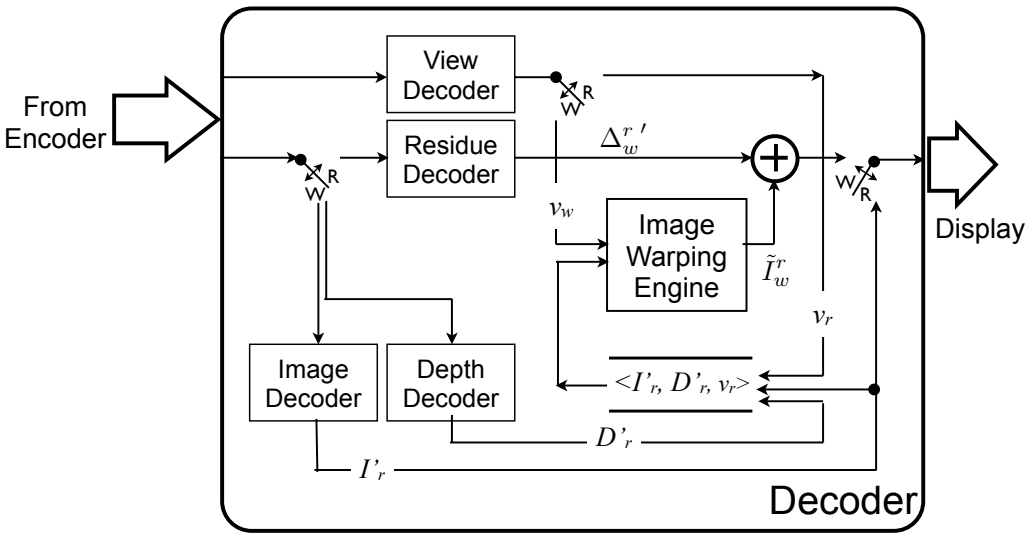


Figure 5.3: Framework of the 3D image warping assisted video decoder

for every W frame. The definition of W is a little bit different:

$$W_x^{v'_{ref(x)} \to v_x} = warping(< I'_{ref(x)}, D'_{ref(x)} >, v_{ref(x)} \to v_x)$$

because $I'_{ref(x)}$ and $D'_{ref(x)}$ are used here to denote the distorted version of passing the original $I_{ref(x)}$ and $D_{ref(x)}$ through both encoder and decoder ($v_{ref(x)}$ is not distorted because we always apply lossless encoding for viewpoints). $ref(x)$ returns the reference R frame for I_x. Since the application scenario is real-time video coding, any frame can only reference from previous frames. Thus $ref(x) < x$. Then we calculate the difference between the warping results and the original video frames as the warping residue $\{\Delta_x^{v'_{ref(x)} \to v_x} | x \in \mathcal{W}\}$, where

$$\Delta_x^{v'_{ref(x)} \to v_x} = I_x - W_x^{v'_{ref(x)} \to v_x}$$

Finally, we encode the video sequence using the depth image of all R frames $\{< I_x, D_x > | x \in \mathcal{R}\}$, residues of all W frames $\{\Delta_x | x \in \mathcal{W}\}$ (Δ_x is used as the short for $\Delta_x^{v'_{ref(x)} \to v_x}$), and all viewpoint information $\{v_x | x \in \mathcal{S}\}$.

On the decoder side, if the received video frame is R frame, we are able to decode I'_r, D'_r and v_r. I'_r should be directly displayed on the mobile screen and at the same time saved in the buffer together with D'_r and v_r. If the video frame is W frame, we get the distorted residue Δ'_w and the viewpoint v_w. We run 3D image warping algorithm for the saved R frame to calculate the warping frame $W_w^{v'_r \to v_w}$, and then retrieve

the target image frame I'_w by adding Δ'_w to $W_w^{v'_r \to v_w}$.

5.3.3 Rate Allocation

The motivation of using 3D image warping in video coding is to reduce the signals of W frames so that they can be more efficiently encoded. The saved bit rate is applied to encode the more important R frames. Here I introduce the rate allocation strategy used in the coding method.

We first analyze the relationships between the different video bit rate components. We can represent the overall rate $r_{\mathcal{S}}$ as follows:

$$r_{\mathcal{S}} = r_{R_I} + r_{R_D} + r_W \tag{5.1}$$

where

$$r_{R_I} = \frac{\sum_{x \in \mathcal{R}} b_{I_x}}{t_{\mathcal{S}}} \tag{5.2}$$

$$r_{R_D} = \frac{\sum_{x \in \mathcal{R}} b_{D_x}}{t_{\mathcal{S}}} \tag{5.3}$$

$$r_W = \frac{\sum_{x \in \mathcal{W}} b_{\Delta_x}}{t_{\mathcal{S}}} \tag{5.4}$$

We did not consider the rate for encoding viewpoints in Eq. (5.1) because the rate used for encoding viewpoint vectors (36 bytes per frame before compression) is neglectable compared with the rate used for image frame compression. Fortunately, x264 allows us to set a target bit rate req_r when encoding a video sequence, and it automatically adjusts the encoding parameters to meet the requirement.

$$req_{R_I} \approx \frac{\sum_{x \in \mathcal{R}} b_{I_x}}{t_{\mathcal{R}}} \tag{5.5}$$

$$req_{R_D} \approx \frac{\sum_{x \in \mathcal{R}} b_{D_x}}{t_{\mathcal{R}}} \tag{5.6}$$

$$req_W \approx \frac{\sum_{x \in \mathcal{W}} b_{\Delta_x}}{t_{\mathcal{W}}} \tag{5.7}$$

Therefore, we do not need to manage the encoding size of every single frame but just find the appropriate bit rate req_{R_I}, req_{R_D}, and req_W to configure x264. We can apply Eq. (5.5), (5.6), (5.7) to Eq. (5.1):

$$r_{\mathcal{S}} \approx \frac{\|R\| \cdot (req_{R_I} + req_{R_D}) + \|W\| \cdot req_W}{\|R\| + \|W\|} \tag{5.8}$$

Currently, we are using a static strategy in the rate allocation. We allocate a fixed portion of the overall available bit rate $f_R \cdot r_{\mathcal{S}}$ to R frames, where $0 < f_R < 1$. We run experiments for each f_R value and

find that 0.5 is a favorable value. The bit rate allocated for R frame depth map encoding is the half of the bit rate allocated for color map encoding because the depth map is not affected by image textures. In practice, we also find that depth encoding can achieve very high quality (50+ dB) with a relatively low bit rate (600Kbps). Therefore, we set a threshold T_{depth} for depth encoding to allocate no more bit rate than T_{depth}. Considering that we run x264 separately for three different components and the difference between the request bit rate and the actual encoded bit rate may be accumulated, we dynamically change req_W based on the actual bit rate of R frame encoding. As a result, given a target bit rate $req_{\mathcal{S}}$, the bit rates of different components are calculated as follows:

$$req_{R_D} = \min(T_{depth}, \frac{\|R\| + \|W\|}{3 \cdot \|R\|} \cdot f_R \cdot req_{\mathcal{S}}) \tag{5.9}$$

$$req_{R_I} = \frac{\|R\| + \|W\|}{\|R\|} \cdot f_R \cdot req_{\mathcal{S}} - req_{R_D} \tag{5.10}$$

$$req_W = req_{\mathcal{S}} + \frac{\|R\|}{\|W\|} \cdot (req_{\mathcal{S}} - r_{R_D} - r_{R_I}) \tag{5.11}$$

5.3.4 Frame Selection

The rate allocation strategy is based on the assumption that the warping residues of W frames contain much less signals and can be encoded more efficiently than original image frames. However, this assumption may not be true if R frames are not carefully selected. In this subsection, we introduce three different frame selection strategies.

Fixed Interval

The *fixed interval* frame selection is the most intuitive solution. Starting from the first frame of the video sequence, we select the frames sequentially to form groups. All frame groups have the same fixed size, which is defined as *warping interval*. The first frame of each group is selected as R frame and the rest are W frames. The R frame in the group is referenced by all W frames of the same group. As long as the *warping interval* remains small, the viewpoints of the frames in the same group are likely to be close to each other so that 3D image warping can help remove most pixels.

The *fixed interval* solution is easy to implement. It does not require any other graphics rendering contexts except the rendering viewpoint and pixel depth required by 3D image warping. The rate allocation for *fixed interval* is also simplified. We do not need to dynamically change the bit rate request because the ratio of R and W is fixed all the time.

Dynamic Interval

One big disadvantage of the *fixed interval* solution is that it is too conservative in selecting W frames. For example, if the virtual camera remains static, all the frames will have the same background scene. Using only one R frame is enough for the whole static sequence. However, the *fixed interval* solution keeps generate R frames every *warping interval*. Toward this issue, we propose a *dynamic interval* strategy. The *dynamic interval* approach processes the encoding in the same way as *fixed interval*, with only one difference. The encoder needs to compare the viewpoint of the currently processing frame with the viewpoint of the previously encoded R frame. If two viewpoints are identical, which means the virtual camera remains static, then the current frame is selected as W frame.

The major benefit of this optimization is that the R frame number can be significantly reduced if the video sequence has a lot of static scenes. The reduction of R frame number allows the rate allocation module in our encoder to allocate more bits for R frame encoding (Eq. (5.9), (5.10)).

Double Warping

This approach uses the same strategy with *dynamic interval* for static sequences, and adds new optimization techniques for motion sequences. As we have discussed in previous chapters, warping artifacts can be effectively removed by compositing the warping results from multiple depth images. Figure 5.4 shows another example of warping two depth images (*double warping*) to compensate holes in a camera panning sequence. According to the figure, if the target viewpoint v_2 is on the right side of the source viewpoint v_1, the viewpoint v_3 of the second reference frame should be selected on the right side of v_2 to provide the best coverage. However, in the scenario of remote rendering, when the virtual camera is panning right, the frame I_3 is actually rendered later than I_2, which means when I_2 is encoded, there is no I_3 available for *double warping* reference.

In order to solve this problem, we modify in the game application used for remote rendering to render auxiliary frames for *double warping*. Figure 5.5 shows the work flow of *double warping* in details. We elaborate the whole flow in an example shown in Figure 5.6. Initially, the viewpoint is at v_1 and the image frame I_1 is captured. If a panning right motion is detected, the encoder will not only encode the current frame I_1, but also request the game program to render the frame I_3 at the viewpoint v_3. I_3 does not exist in the game video sequence, but is generated only to support *double warping* for all intermediate viewpoints between v_1 and v_3. Both I_1 and I_3 are selected as R frames and saved in the buffer. As time goes by, the viewpoint pans right to v_2. It is well covered by two R frames I_1 and I_3. Thus I_2 is selected as W frames and *double warping* is applied to calculate the residue. If the viewpoint keeps moving to v_4, which is out of

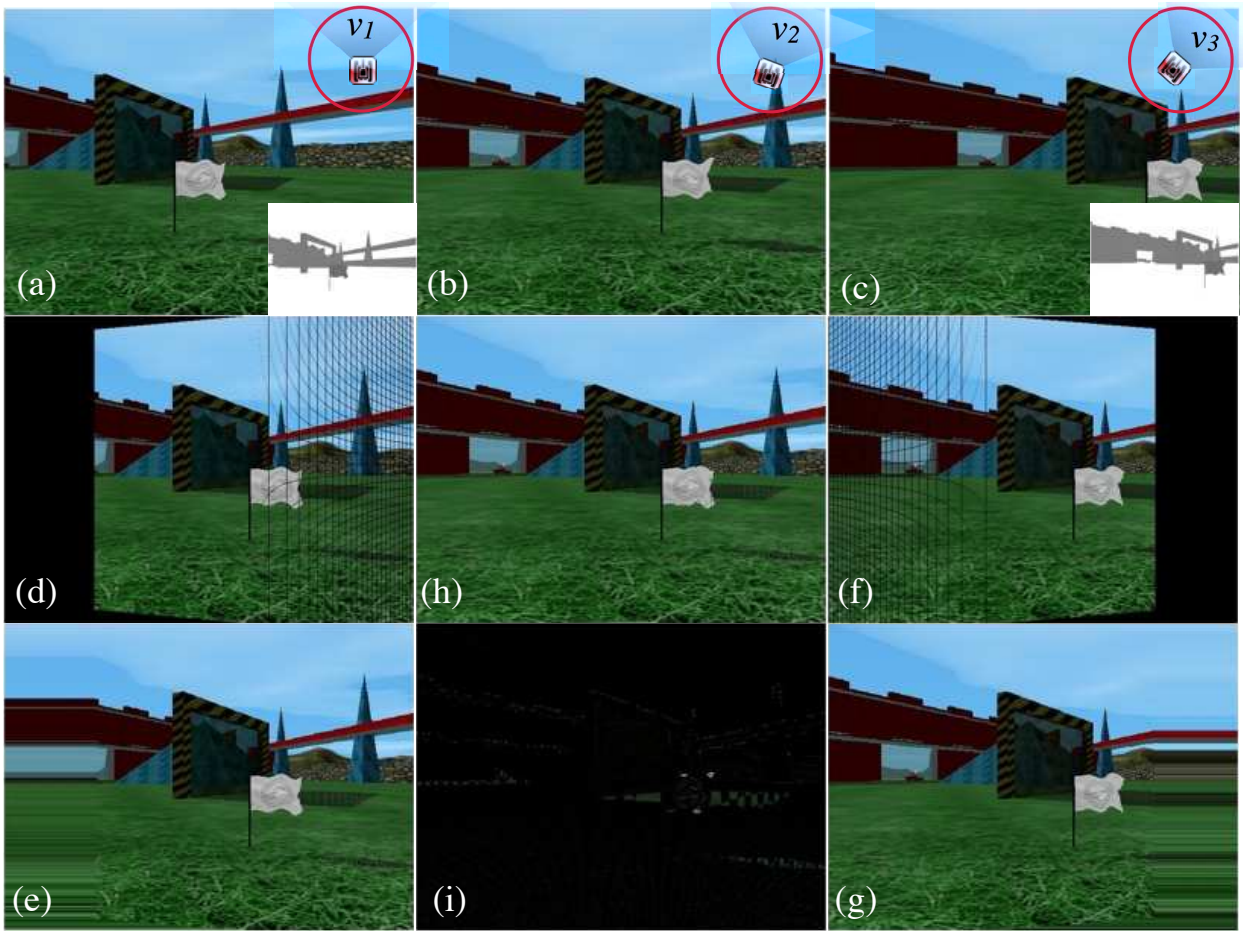


Figure 5.4: An illustration of how double warping, and hole filling work: (a) the depth image frame $< I_1, D_1 >$ at viewpoint v_1; (b) the image frame I_2 at viewpoint v_2; (c) the depth image frame $< I_3, D_3 >$ at viewpoint v_3; (d) $W_2^{v_1' \to v_2}$ without hole filling; (e) $W_2^{v_1' \to v_2}$ with hole filling; (f) $W_3^{v_1' \to v_3}$ without hole filling; (g) $W_3^{v_1' \to v_3}$ with hole filling; (h) the warping result of double warping $W_2^{v_1' \to v_2} \bigcup W_3^{v_1' \to v_3}$, with hole filling; (i) the difference between (h) and (b)

the coverage area of I_1 and I_3, the encoder will ask the game engine to render a new auxiliary frame I_5 at the viewpoint v_5. I_5 will be selected as R frame, added to the buffer to replace I_1. Both I_3 and I_5 are used to support the double warping of I_4.

Compared with the previous two frame selection strategies, *double warping* is able to improve the encoding performance further by using fewer R frames and reducing the warping residues created in the motion sequence. *Double warping* not only requires the rendering viewpoint and pixel depth for 3D image warping, but also detects the camera motion events in the 3D video game engine and reuses the rendering engine to generate auxiliary frames.

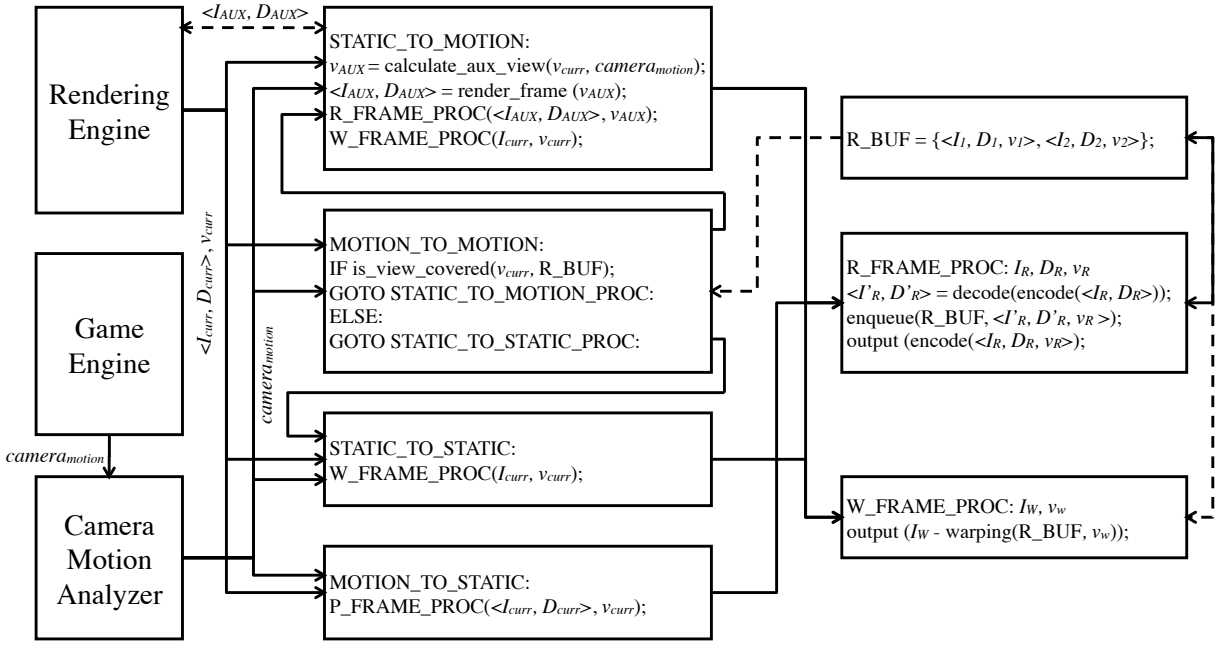


Figure 5.5: *Double warping* frame selection

5.4 Implementation

We have implemented an off-line version of the proposed video coder. In this section, we discuss some implementation issues and share some lessons we have learned.

The latest version of x264 is used in our system to encode the image map, depth map, and warping residues. We configure x264 as a real-time encoder to meet the requirement of our cloud gaming system. Table 5.1 lists the parameters for a real-time configuration.

Table 5.1: Real-Time x264 Encoding Settings

Setting	Description
`--bframes 0`	No bidirectional prediction
`--rc-lookahead 0`	No frame type and rate control lookahead
`--sync-lookahead 0`	No threaded frame lookahead
`--force-cfr`	Constant frame rate timestamp
`--sliced-threads`	Sliced-based threading model
`--tune zerolatency`	Implies all above real-time settings
`--preset fast`	A good balance between speed and quality
`--keyint 4`	Use small GOP size

The encoding of depth and residue is quite tricky in our current implementation. Our encoder produces the depth map as a 16-bit grey scale image. Because the current x264 does not natively support 16-bit

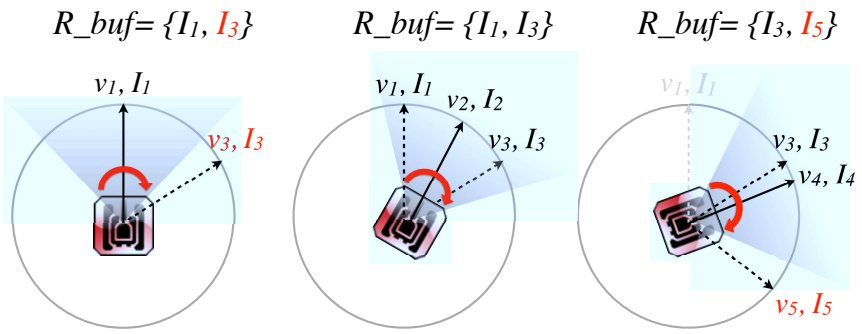


Figure 5.6: An illustration of the generation of auxiliary frames

color, it automatically converts the depth map to 8-bit image before encoding it, which leads to huge quality degradation. Therefore, we decompose the 16-bit depth map into two 8-bit sub-images: D^H and D^L. D^H contains the 8 more significant bits of every pixel and D^L has the 8 less significant bits. Then we encode D^H with the lossless mode of x264 by setting `--qp = 0`. D^L is encoded in a similar method to the image map. The residue frame can not be directly encoded by x264 either because it is a 9-bit color image (the difference of two 8-bit color image). Although x264 has the 9-bit support, we have found that in practice, the 9-bit x264 encoding does not generate good quality results and expensive operations are needed to prepare the 9-bit residue frame for the x264 input. Hence, we are taking an alternative approach by creating a new 8-bit image Δ^{+-} with the same width but twice height. The upper part of Δ^{+-} stores the pixels with positive value in Δ and the lower part has the negative pixels (reversed values). We use x264 to encode Δ^{+-} with the same parameters for image map and depth map.

We apply a hole filling technique in 3D image warping to fill all the holes with the color of their nearest neighbors (Figure 5.4 shows two examples). In the implementation, we found out that the hole filling is actually very important to our video coding method because the H.264/AVC encoder cannot effectively deal with the warping residues caused by hole pixels. It usually results in using even more bits to encode the residue of one hole pixel than the whole original image block. Other hole filling techniques proposed in [43, 3] can also be tried.

We set the GOP size for R frame encoding to 1 to force that every R frame is encoded using intra-prediction. This is because in the situation of mobile cloud gaming, the mobile networks are less likely to provide reliable end-to-end transmission. We should set the GOP to a small number so that the system does not take too long to read the next I frame if an I frame is lost. In our work, all W frames are dependent on their reference R frame. Thus, a larger than 1 GOP size for R frame encoding can possibly be scaled up to a huge loss when the network drops an intra-predicted R packets. Even though all R frames are encoded with intra-prediction, our encoder can still suffer when the R frame of a long static sequence is lost. We consider

Table 5.2: Default Encoding Settings

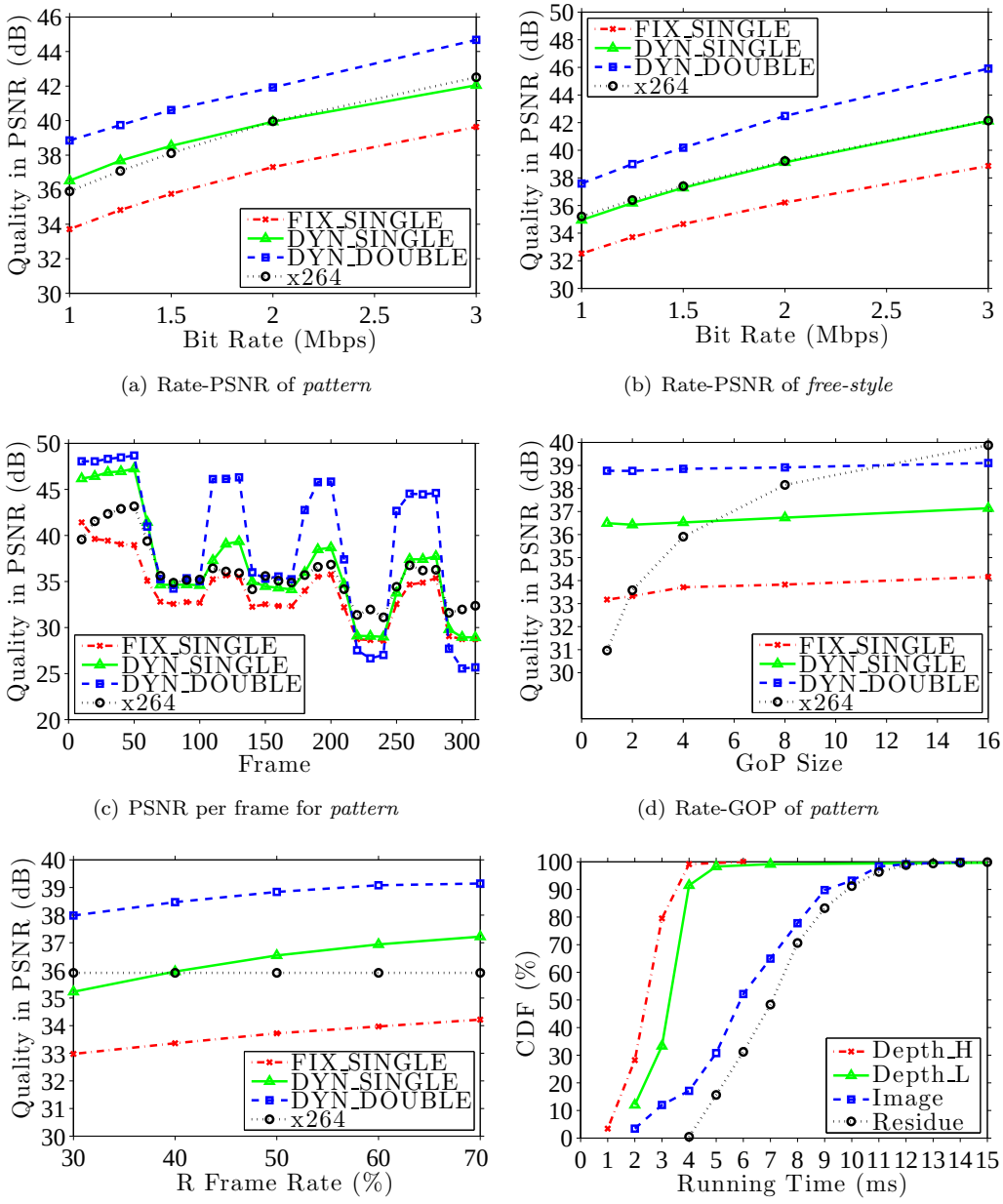
Setting	Description
$req_{\mathcal{S}} = 1$Mbps	The target video bit rate
$f_R = 0.5$	Half bit rate is allocated for R frames
$T_{depth} = 700$	The threshold for depth encoding
`R_GOP` $= 1$	GOP for R frame encoding
`W_GOP` $= 4$	GOP for W frame residue encoding
`warp_int` $= 4$	*Warping interval* used only by the *fixed interval* and *dynamic interval* approaches

solving this problem in the future work by integrating different reliable transmission approaches.

5.5 Evaluation

In this section, we evaluate the performance of our proposed 3D image warping assisted video coding with the prototype remote rendering system introduced in Chapter 3. *BZFlag* game is rendered on the server. Since we currently only have implemented an off-line version encoder, we need to record the game video first and then pass the raw frames to our encoder. We captured two game video sequences: *pattern* and *free-style*. *Pattern* contains 315 frames, which is a 10 second play with four different motions (pan left, pan right, move forward, and move backward) currently supported by the mobile client implemented by us. Four motions are all played in the same pattern: pause for about one second and then move for about one second. *Free-style* contains 4072 frames, which is a 2 minute 15 second video captured during the actual game play. We encode two video sequences with three different scripts: *fix_single*, *dyn_single*, and *dyn_double*, which stand for three different frame selection strategies: *fixed interval*, *dynamic interval*, and *double warping*, respectively. Table 5.2 lists the default setting for all scripts. We compare the results with the performance of using x264 directly for real-time encoding. The x264 encoding is configured using the real-time setting in Table 5.1. The default target bit rate is also set to 1Mbps. We present all experimental results in Figure 5.7.

The rate-distortion curves are presented in Figures 5.7(a) and 5.7(b). Both figures indicate that *dyn_double*, which stands for *double warping* optimization, has the best performance of all three approaches. It also outperforms x264 for about 2-3 dB at any given rate. In Figure 5.7(c), we plot the PSNR value of every frame in sequence *pattern* and we can see how each encoding approach actually works. The line of *dyn_double* shows periodic and prominent rises in PSNR because *double warping* uses the least number of R frames, and allocates the highest rate to encode R frames. Therefore all frames in the static scene can benefit a lot. However, since less bit rates are allocated for W frames, the PSNR of motion frames drop fast. For the first



(a) Rate-PSNR of *pattern*

(b) Rate-PSNR of *free-style*

(c) PSNR per frame for *pattern*

(d) Rate-GOP of *pattern*

(e) Rate-f_R of *pattern*

(f) Encoding Runtime

Figure 5.7: Experiment Results

two motions (pan left and pan right), the frame quality can still maintain in a high level because *double warping* can generate high quality warping results. For the last two motions (move forward and move backward), the quality drops significantly because *double warping* can not correctly handle some special graphical effects, such as shadows. Figure 5.8 is the frame number 300 (PSNR=24.35dB) in *pattern* sequence using *dyn_double*. It shows that that the major problem comes from the shadow on front ground.

Figure 5.8: Shadow problem for *double warping*

We also compare the performance under different setting parameters. For example, Figure 5.7(d) studies the relationship between PSNR and the GOP size. The figure shows no obvious difference in encoding quality of the proposed approaches with different GOP settings. However, the performance of x264 encoding increases significantly as the GOP size increases, and can beat our best performance (*dyn_double)* when GOP size is 16. We also test different f_R in Figure 5.7(e). Considering the line (*dyn_double*) having the best performance, the increase is less obvious for $f_R > 0.5$. Thus, we keep selecting $f_R = 0.5$ for all our experiments because assigning less bit rate for W frame can deteriorate the frames with bad quality although the average PSNR may increase.

Finally, the encoding and decoding complexity of the proposed coding method is evaluated. For video encoding, each W frame runs 3D image warping (either single warping or double warping) and the encoding of Δ. Each R frame needs to encode and decode I, D^H, and D^L. We plot the runtime of all components in Figure 5.7(f) and find that 90% of D^H and D^L frames are encoded within 4 ms, and 90% of I, and Δ frames are encoded within 10 ms (experiments were run on a Linux server with an Intel i7 2.8Ghz CPU and 4GB memory). Therefore, we believe there is enough room to optimize for 30 fps real-time encoding. For video

decoding, we compare the overheads of our coding method with x264 decoding. Each R frame decodes I, D^H, and D^L and each W frame runs 3D image warping and decodes Δ. Assuming the decoding time of Δ and I is comparable to the decoding time of the original image frame, the added complexity of our method over x264 is the decoding of I for auxiliary frames (a subset of R frames), the decoding of D^H, D^L for all R frames, and the 3D image warping for every frame. Our experiments indicate that usually less than 10% of all frames are selected as R frames and the 3D image warping operation can be optimized to as low as 1.7 ms per frame [93]. Therefore, we consider the overheads are acceptable. In addition, since our scheme is on top of H.264, the decoder can take advantage of the hardware acceleration module. The extra memory operations should not be a bottleneck as mobile devices become more and more powerful.

5.6 Discussion

The 3D image warping assisted video coding method can be easily integrated with other modules in the proposed remote rendering system:

1. The depth image references generated for latency reduction can simply be encoded as R frames using this coding method. When there is a camera motion, the reference that covers this motion can be directly used to warp W frames. The coder only needs to adjust f_R to allocate enough bits for the "extra" R frames.

2. The *double warping* can be easily implemented in RRK. The design of RRK already has the APIs to request the rendering program to render extra frames as multiple references, adding the function to render auxiliary R frames will be straightforward.

3. The reference selection algorithms, especially the *Reference Prediction Algorithm* introduced in Chapter 4 can be reused in the coder to decide what is the best viewpoint to generate an auxiliary R frame.

4. The auxiliary R frames can also be regarded as a reference.

There are also limitations of our proposed video coding method. First, it only works with cloud gaming systems or other remote rendering systems where the video encoder can extract graphics rendering contexts from the rendering engine in real-time. It may not achieve the best performance in the motion intensive games rendered in the third person perspective (e.g., real-time strategy and sports games) because the motion in the video is not mainly caused by the movement of virtual cameras. Second, the coding performance of the proposed video coding method drops fast if the majority of pixels in the image frames are animations, foreground object movements, or special rendering effects (e.g., shadows). This is because the proposed

video coding method relies on the warping residues to encode those pixels, which are allocated relatively less bits to encode. Even though, we believe our coding method can still be attractive to cloud gaming service providers. This is mainly because the first person perspective games usually require more intensive graphics rendering and are more suitable for cloud gaming. For example, according to the current game catalog of OnLive, more than half are the first person shooting, action, or racing games that can potentially benefit from our coding scheme. In order to extend the proposed coding method for all games, we will pursue to integrate the proposed warping assisted coding scheme as a coding mode into x264 (similar to intra/inter mode), so that certain graphic effects, regions, and games can be encoded using traditional H.264 standard.

5.7 Summary

I have introduced the concept of using graphics rendering contexts in real-time video coding. Different from the conventional video coding approach, my coder takes advantage of the pixel depth, rendering viewpoints, camera motion patterns, and even the auxiliary frames that do not actually exist in the video sequence to assist video coding. I have proved that the approach of integrating more components in the system has the potential to outperform the state-of-art H.264/AVC coding tools in the real-time cloud gaming scenario.

The proposed coding method is novel in applying the rendering contexts and 3D image warping to assist real-time video coding. Many concepts have been proposed by earlier researchers in different scenarios. For example, Levoy [38] proposed to use polygons to assist JPEG/MPEG compression. 3D image warping has been similarly applied for different purposes in various remote rendering systems [43, 3, 25] and other applications [66, 50]. However, I believe I am the first to study using the rendering contexts to assist real-time video coding in the area of remote rendering and cloud gaming, and I present the best solution so far to integrate the graphics rendering contexts and IBR techniques into video coding.

Chapter 6

Interactive Performance Monitor

There remains the last component of the proposed remote rendering system, performance monitor. Considering there have already been mature tools to monitor many aspects of system performance (such as network bandwidth, CPU usage, etc.), my research only focuses on monitoring the interactive performance of the remote rendering system.

Conventionally, latency is the metric to evaluate the interactive performance of the rendering system. However, since my proposed remote rendering system can reduce the interaction latency by synthesizing frames on the client, using latency only can no longer accurately measure the interactive performance of the system. Therefore, a new metric called DOL (*distortion over latency*) is developed to work with a novel run-time interactive performance monitor. In this chapter, the design details as well as the experiment results of the real-device evaluation will be presented.

6.1 Introduction

The performance monitor module examines the information of system execution at run-time and adjusts different parameters to adapt the rendering system to the available computing and networking resources. Here we do not want to talk about how to detect network round trip time, available bandwidth, or CPU usage because there have been numerous tools and methods developed for these purposes (e.g., [22]). Instead, we focus on monitoring the interactive performance of the remote rendering system.

The interactive performance indicates how the system reposes to the user interactions. In the proposed remote rendering system, there are several parameters directly related to the system interactive performance. For example, f_R in Section 5.3.3 determines how to allocate rate for R frame encoding. The parameter should be set to a small value when the user does not interact much with the application and most frames are encoded as W frames due to the unchanged rendering viewpoint. If the user changes viewpoint frequently, more AUX frames are generated and encoded as R frames. In this case, f_R should be set to a large value to guarantee the quality of R encoding. Another example is the $psnr_{resp}$ used in reference selection algorithms (Chapter

4). When the camera motion becomes intensive, a smaller $psnr_{resp}$ is demanded to generate the references covering a larger *reference range*. These parameters can only be adjusted by the performance monitor based on the user behaviors at run-time and the system interactive performance.

Traditionally, the interactive performance of a rendering system is evaluated by *latency* [19, 72, 90, 35, 12], which is defined as the time from the generation of user interaction request till the appearance of response image, same as the term *interaction latency* we used in this thesis. However, the metric is not sufficient to evaluate the real interactive performance of the remote rendering system proposed in this thesis. Since the mobile client is able to run 3D image warping algorithm to synthesize the image for the updated viewpoint based on the received depth image references, it can respond immediately to any viewpoint change user interaction. This *post rendering* technique reduces the *interaction latency* at the cost of increasing network bandwidth usage and degrading rendering quality (regarding to the quality of the warping results), so that *interaction latency* is no longer an effective performance metric. Instead, we need a new metric to combine both latency and quality factors to measure the interactive performance.

There have been many related researches proposed in the history. Endo *et al.* [19] discovered two factors of an interactive performance: *interaction latency* and *throughput*. Salzmann [69] studied combining multiple parameters as a metric for QoS in real-time interaction over internet. Lamberti and Sanna [35] discussed in depth about different factors that affect *interaction latency* in remote rendering. Some projects [43][27][78] also studied the issue of rendering quality. In the previous chapters, we also used the metrics such as *warping error* or *reference range* for performance evaluation.

In order to effectively monitor and compare the interactive performance of the proposed remote rendering system, I propose a new interactive performance metric: DOL (*distortion over latency*) that integrates both latency and quality. My work is different from previous works in integrating quality together with latency as one metric to measure the interactive performance of remote rendering systems. I will also introduce how the performance monitor can calculate DOL at runtime.

6.2 Distortion Over Latency

The new metric DOL should provide two abilities:

- Measure both the latency of interaction events and the rendering quality on mobile clients.
- Integrate two measurements into one result, the value of which can be used to evaluate interactive performance.

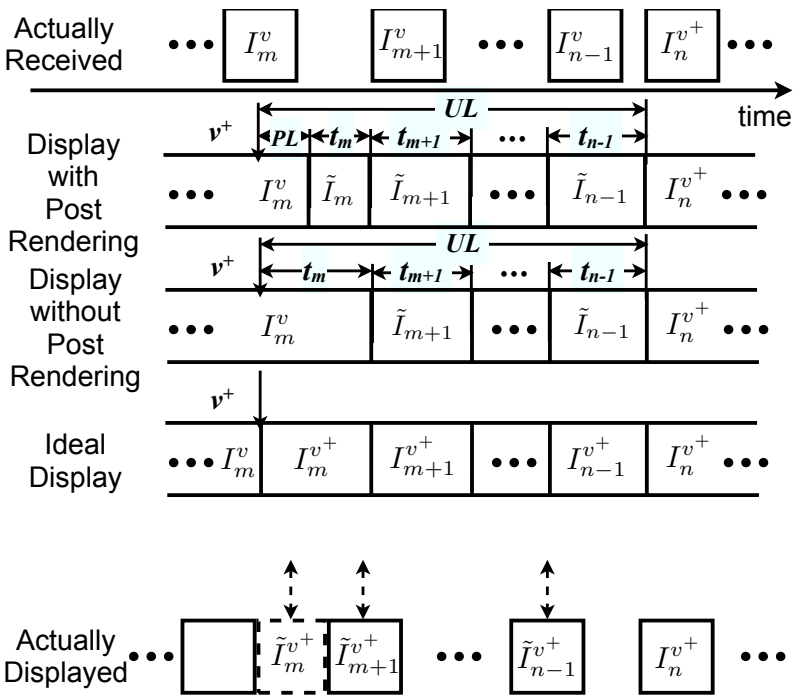


Figure 6.1: Images frames displayed on the mobile client of a remote rendering system. Distortion over latency is the sum of difference between the actually displayed frames and the ideal frames during the interaction latency

We first define several notations and concepts. Figure 6.1 shows an illustration of image frames received and displayed on the mobile client in time sequence. I_i^v $(m \leq i \leq n-1)$ denotes the frame i which is rendered at viewpoint v on the rendering server. The user interaction that changes the viewpoint from v to v^+ happens right after the display of frame m. Frame n is the first frame that is rendered at v^+ on the rendering server. $\tilde{I}_i$ denotes the frames that are actually displayed on the mobile client. Since the metric is not developed to server our own system, we consider both the systems with and without post-rendering module (3D image warping). If the mobile client has a post rendering module, $\tilde{I}_i$ is the result frame of re-rendering I_i^v to the new viewpoint v^+. In our case, $\bigcup_k W_i^{ref_k \to v^+}$ is displayed Otherwise, $\tilde{I}_i$ is identical to I_i^v.

We define *frame interval* t_i $(m+1 \leq i \leq n-1)$ as the time interval that frame i stays on the screen. Note that the *frame interval* t_m is defined in a different manner. For a post-rendering system, t_m is the time interval that $\tilde{I}_m$ appears on the screen. For a non-post-rendering system, t_m is the time from the moment of user interaction till the appearance of $\tilde{I}_{m+1}$. *Post-rendering latency* PL is defined as the time from the moment of user interaction till the appearance of $\tilde{I}_m$. If the mobile client does not have any post rendering module, then $PL = 0$. *Update latency* UL is defined as the time from the moment of user interaction till

the appearance $I_n^{v^+}$. It can also be expressed as:

$$UL = PL + \sum_{x=m}^{n-1} t_x \tag{6.1}$$

According to the traditional definition of *interaction latency*, PL is the *interaction latency* for post-rendering systems and UL is the *interaction latency* for non-post-rendering systems.

Rendering quality is determined by calculating the distortion between the actually displayed $\tilde{I}_i$ and the corresponding ideal frame $I_i^{v^+}$, which is rendered at the correct viewpoint. The ideal frame is introduced for distortion calculation only and it is not actually displayed or generated. The distortion between two image frames is calculated by *mean square error* as follows:

$$D(\tilde{I}_i, I_i^{v^+}) = \frac{\sum_{x=1}^{h} \sum_{y=1}^{w} [\tilde{I}_i(x,y) - I_i^{v^+}(x,y)]^2}{w \cdot h} \tag{6.2}$$

where w and h are the width and height of the image, and $I(x,y)$ is the intensity value of pixel (x,y) in the image frame I.

We now define DOL (*Distortion Over Lancy*) as the total distortion of all image frames displayed during the *update latency* UL.

$$DOL = 10 \log_{10} \frac{MAX_I^2 \cdot UL}{PL \cdot D(I_m^v, I_m^{v^+}) + \sum_{i=m}^{n-1} t_i \cdot D(\tilde{I}_i, I_i^{v^+})} \tag{6.3}$$

where MAX_I is the maximum possible pixel value of the image (e.g., 255 for 8-bit images). DOL is expressed in terms of the logarithmic decibel scale so that the unit is dB.

According to Eq 6.3, the DOL score is determined by both latency and the quality of displayed images. Either reducing the latency or increasing the post rendering quality leads to high DOL scores. Therefore, we believe that the higher DOL score represents the better interactive performance. Since DOL is normalized with image resolution, pixel value, and frame rate, the score can also be used to compare the interactive performance of rendering systems with different configurations.

6.3 Run-time DOL Monitor

DOL can be easily calculated offline by saving all system execution profiles. The difficulty of online monitoring is that the mobile client does not have $I_i^{v^+}$ at runtime to calculate the distortion. In this section, we

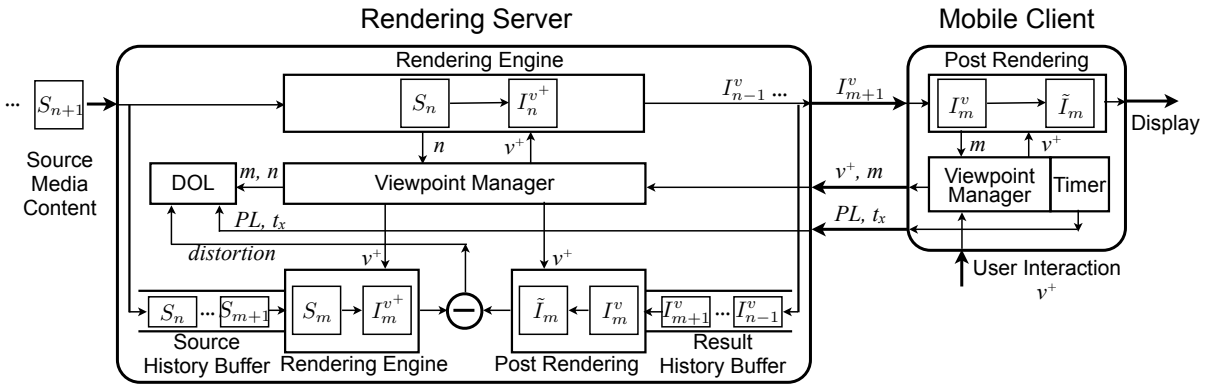


Figure 6.2: Framework of the remote rendering system with online interactive performance monitor

propose a light-weight scheme for our performance monitor to simulate all behaviors of the mobile client on the rendering server and then calculate DOL at runtime. Figure 6.2 explains the framework of the proposed run-time monitor and how data flows. Note that all the highlighted components belong to the performance monitor module shown in Figure 3.6.

6.3.1 Client

The viewpoint manager needs to check for the current image frame number (m in Eq. 6.3) when user interaction happens. The frame number together with the new viewpoint v^+ are sent back to the rendering server. In addition, a timer component should be added to record the *post-rendering latency* PL and all *frame interval* t_x and send back to the rendering server. The timer starts on the arrival of the user interaction that changes v to v^+, records the elapsed ticks every time when the mobile screen is updated, and stops when the first frame $I_n^{v^+}$ is displayed. However, in our implementation we notice that the v^+ (a set of float vectors to indicate camera position, camera direction and camera up direction) on the mobile client can be slightly different from the v^+ processed on the rendering server due to the difference of floating point precision, and the difference can keep the timer running forever. In order to resolve this problem, we introduce the concept of *viewpoint age*. A 32-bit unsigned integer is assigned as the "age" to every viewpoint. The *viewpoint age* increases upon user interaction. Every image frame inherits the *viewpoint age* from its rendering viewpoint. Therefore, the timer module compares only the integer *viewpoint age* of the current mobile viewpoint and the received frame rather than the float vectors.

6.3.2 Server

Two history buffers are added on the rendering server to store both source content frames and rendering results frames. The buffer size is determined by the maximum possible *update latency* and the rendering frame rate. For example, if rendering frame rate is 10 fps and the maximum possible *update latency* is 2 seconds, the buffer should store no less than 20 frames. Two buffers are organized as FIFO queues. The source history buffer is connected to a rendering engine and the result history buffer is connected to a post rendering module identical to the one on the mobile client (if the mobile client does not have post rendering support, the result history buffer outputs the saved frames directly). Under the control of viewpoint manager, the source history buffer can generate the history frame rendered at the correct viewpoint ($I_x^{v^+}$) and the result history buffer can generate the history frame exactly displayed on the mobile client ($\tilde{I}_x$). The distortion of these two frames are calculated and sent to the DOL module. The viewpoint manager initiates the DOL calculation when the new viewpoint information and the frame number (m in Eq. 6.3) arrive. Meanwhile, it checks the current frame number (n in Eq. 6.3) in the rendering engine because it is the first frame rendered at the updated new viewpoint. Both frame numbers (m, n) together with PL and t_x are passed to the DOL module to calculate the DOL score at runtime.

Our runtime DOL calculation scheme is light-weight for the mobile client and network usage. Neither timer recording nor the network transmission of extra data can cause noticeable performance degradation. The added components and functions on the rendering server may double the computation workload. However, given the assumption that the rendering server has abundant computing resources, our scheme should not impair the rendering performance of the server. In the worst case, all DOL calculation related components can be deployed in an individual workstation so that the server for rendering is not affected.

6.4 Evaluation

In this section, we demonstrate that DOL, although looks simple, actually works well. Our experiments try to prove that given two rendering systems, the one with higher DOL score has better interactive performance. We set up a remote rendering system with different configurations which result in different interactive performance, run experiments for all configurations, and compare the collected DOL scores with the actual performance.

Experiments are running on remote rendering system prototype introduced in Section 3.4. A 3D image warping module has been implemented to provide post rendering support. The prototype can work in three different rendering modes:

- *Image Streaming*: the rendering server sends only the 2D image of the rendering result. The client simply displays the received image on the screen. The interactive performance of this rendering mode degrades as the network delay increases.

- *Single Warping*: the rendering server sends one depth image (including color map and depth map) per frame. The client displays the color map if the received frame is rendered at the correct viewpoint. Otherwise, the client warps the depth image to the correct viewpoint using 3D warping algorithm. This rendering mode uses the synthesized images which have warping holes (Figure 3.3) to compensate the interaction latency. It is expected to have better interactive performance than *image streaming* when using the same network.

- *Double Warping*: the rendering server generates two carefully selected depth images per frame based on the reference selection algorithms proposed in Chapter 4. The client warps both depth images to the current rendering viewpoint if neither can be displayed directly. This rendering mode can generate the warping results with less holes (Figure 3.5) and should have the best interactive performance of all three modes when using the same network.

Given the same network condition, the interactive performance the three rendering modes can be ranked as: *Double warping* > *Single warping* > *Image streaming*[1].

The rendering server runs on a workstation that has an AMD Phenom II 3.2GHz quad-core CPU, 4GB memory, an Nvidia GeForce 9800GT GPU, and connects to the Gbps ethernet university network. The mobile client has been implemented to run on Apple iOS platforms. Three different setups are selected to run all experiments:

- *Simulator*: the mobile client runs on an iPhone simulator hosted by an Apple MacBook laptop. The host machine has a dual-core CPU, 2GB memory, and connects to the university network through 802.11n Wi-Fi.

- *Wi-Fi*: the mobile client runs on an Apple iPhone 4 smartphone. The phone connects to the university network through 802.11g Wi-Fi.

- *3G*: the mobile client runs on an Apple iPhone 4 smartphone. The phone connects to Internet through 3G (HSDPA/UMTS) network operated by AT&T.

Table 6.1 shows the SpeedTest[2] results of the network used in three setups above. *Ping* denotes the network roundtrip time between server and client. *Download* and *upload* characterize the practical network bandwidth

[1] $A > B$ means A leads to better interactive performance than B
[2] http://www.speedtest.net

Table 6.1: SpeedTest Results of Three Network Setups

	Simulator	*Wi-Fi*	*3G*
Ping (ms)	8	43	248
Download (Mbps)	61.75	9.91	2.02
Upload (Mbps)	7.85	7.73	0.71

available for communication. The network with short roundtrip time and high bandwidth is expected to deliver better interactive performance than the network with long roundtrip time and low bandwidth. Thus, we can rank these setups as: *Simulator* > *Wi-Fi* > *3G*.

Two applications *bunny* and *taichi* are used for experiments. Both applications are rendered on the server at a fixed resolution of 480×320. Color images are compressed with JPEG and depth images are compressed with ZLIB. TCP protocol is used to guarantee the reliable transmission. The static model *bunny* is either re-rendered every second or updated when any user interaction changes the rendering viewpoint. The 3D video *taichi* is rendered at a constant frame rate of 8 fps with one exception that when the rendering server is on *double warping* mode and the mobile client connects through *3G*, only 5 fps can be achieved due to the limited bandwidth. Table 6.2 lists the actual network bandwidth used in our experiments for different configurations.

Table 6.2: Network Bandwidth (Kbps)

	Image Stream.	*Single Warp.*	*Double Warp.*
Bunny	56	496	992
Taichi	320	1024	2048

For each run of our experiment, we send the same group of user interaction requests to the mobile client with a fixed interval of 10 seconds and last for 10 minutes. The user interaction either translates or rotates the rendering viewpoint. The interaction is controlled by script so that the same request is always sent at the same time for every experiment run.

Table 6.3: Interaction Latency (ms)

		Simulator	*Wi-Fi*	*3G*
Image Streaming	*Bunny*	40	255	449
	Taichi	285	474	610
Single Warping	*Bunny*	17	161	144
	Taichi	16	156	143
Double Warping	*Bunny*	18	177	161
	Taichi	16	154	146

We present the data collected in our experiments in Table 6.3, 6.4, and 6.5 to summarize the average *interaction latency*, the average *update latency*, and average DOL score, respectively. We list *interaction*

Table 6.4: Update Latency (ms)

		Simulator	*Wi-Fi*	*3G*
Image Streaming	*Bunny*	40	255	449
	Taichi	285	474	610
Single Warping	*Bunny*	70	367	847
	Taichi	256	521	746
Double Warping	*Bunny*	122	377	1038
	Taichi	274	536	931

Table 6.5: Distortion Over Latency

		Simulator	*Wi-Fi*	*3G*
Image Streaming	*Bunny*	37.91	29.84	27.39
	Taichi	35.65	33.44	32.34
Single Warping	*Bunny*	38.47	30.25	28.04
	Taichi	42.13	36.60	35.98
Double Warping	*Bunny*	40.00	31.12	30.52
	Taichi	43.37	37.15	36.33

latency in the table to compare with our proposed metric DOL. As we have discussed previously in this chapter, *interaction latency* equals to UL for image streaming mode and equals to RL for other two warping modes.

From the data in the tables, we can find the *interaction latency* works well for the image streaming mode. According to the *interaction latency* numbers in the first row, *Simulator* > *Wi-Fi* > *3G* (the shorter latency means better performance). However, it fails to work for two warping modes. The *interaction latency* numbers in last two rows lead to incorrect ranks: *3G* > *Wi-Fi*, *Single warping* > *Double warping*.

Our proposed DOL does a much better job in ranking all cases. Comparing the DOL scores in every row and column, we can easily conclude that *Simulator* > *Wi-Fi* > *3G*, and *Double warping* > *Single warping* > *Image streaming*. These conclusions perfectly match our experiment expectations.

6.5 Summary

We have introduced how to design a run-time performance monitor to evaluate the interactive performance of a remote rendering system. A new metric DOL is proposed as a better method than *interaction latency* only to accurately measure the interactive performance of the remote rendering system that uses post-rendering techniques to reduce latency.

Although we have not evaluated DOL quantitatively or correlated it with a meaningful scale, the current qualitative research on DOL is already sufficient enough to help the performance monitor to adjust remote rendering parameters. There are many places we can improve the metric in the future. First, MSE is not

the best metric for image quality evaluation. We can test more image assessment tools, like SSIM, for the metric. Second, the human reaction to the *interaction latency* is not linear. For example, the system user is not sensitive to the latency if it is less than a threshold [12]. The next version of DOL should take this fact into consideration. Third, the current study of DOL is still limited in using the metric to measure which system has the better interactive performance. However, a complete evaluation also requires some quantitative studies to correlate the numbers in DOL score with the difference of interactive performance that human can perceive. Thus, more subjective tests should be carried out to better understand the connections between DOL scores and user satisfaction.

Chapter 7

Conclusion

7.1 Achievements

This thesis research started from a simple idea of using the powerful server to improve the 3D graphics rendering experience on mobile devices, and ended with a neat low-latency remote rendering design, a prototype system that runs different applications on real devices. Here I summarize the major achievements and results of my thesis work:

1. I compared different remote rendering technologies and found out that using multiple reference is the most appropriate approach for low-latency rendering on mobile devices.

2. I designed a remote rendering system using the *Multi Depth Image* approach and practiced with a prototype system implementation.

3. I converted the latency reduction into a reference selection problem and proposed several solutions, including the *Reference Prediction Algorithm* that fits the real-time rendering system.

4. I discovered the idea of using graphics rendering engine to generate auxiliary frames for the purpose of encoding other frames more efficiently.

5. I developed a novel real-time video coder, that achieves better rate-quality performance compared with the state-of-art H.264 coding tool in the real-time coding scenario.

6. I proposed a new metric that combines both latency and rendering quality for interactive performance evaluation and tested the metric with different system setups.

7.2 Discussion and Future Work

This thesis work proves that the remote rendering design using multiple depth images can effectively reduce the interaction latency. However, the current solution still has several limitations:

- 3D image warping only works the user interactions that change the rendering viewpoint, but does not help the movements of the foreground objects.
- The image based rendering techniques can not handle special graphical effects correctly, such as shadow effects (Figure 5.8).
- The user interaction that controls the avatar action can not be supported at all using the current remote rendering system.

The future work towards the a better remote rendering design should take more advantage of the increasing graphics rendering capability and networking bandwidth for the mobile client. Although the development of mobile chip in the next decade may not be enough to replace desktop GPU in running complex 3D graphics, it will be able to share more workloads rather than justing running an image based rendering algorithm. The next generation of wireless technologies may not be able to reduce the network latency much, but will surely provide larger bandwidth for the remote rendering system to stream more data in real-time. I believe the following lessons I have learned from this thesis work will be useful for the future research in this area.

- **Computation distribution hides latency**: The idea of creating an image-based representation of the original 3D graphics scene on the server, and using this image-based representation to synthesize high quality display image on the client does not reduce the network latency, but hides the latency with the computation collaboration on both sides. The nature of creating an image-based representation for the original 3D graphics scene is to distribute the rendering computation between the server and the client. For different computation and bandwidth resources, such computation distribution should be adjusted accordingly to fully utilize the resources and achieve the best performance.
- **The context information always helps**. The contexts include not only the information already created and available, but also the information which can be created with the existing sources. However, the second type of context information can easily be omitted because it does not exist. I have shown that creating extra frames can actually improve the video coding performance in this thesis, I believe such concepts will be helpful in many other areas.
- **New programming model for cloud and mobile convergence**. The best remote rendering performance requires the application developers to be involved front he beginning. The developer should be aware of the new server-client architecture and how latency is generated. The development of the remote rendering applications should divide all computing tasks according to their computation and latency requirements, and distribute to the appropriate sites. For example, the client can directly

perform low-computation tasks and the latency insensitive tasks leave on the server. Only the tasks that require low-latency and high-computation need to be specially treated. New programming models and tools will be needed in the future for implementing these ideas.

References

[1] Apple. Cgl reference. http://developer.apple.com/library/mac/documentation/Graphics Imaging/Reference/CGL_OpenGL/CGL_OpenGL.pdf, 2009.

[2] R. T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355 – 385, 1997.

[3] P. Bao and D. Gourlay. Remote walkthrough over mobile networks using 3-d image warping and streaming. *Vision, Image and Signal Processing, IEE Proceedings* -, 151(4):329 – 336, aug. 2004.

[4] R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: a virtual display architecture for thin-client computing. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 277–290, New York, NY, USA, 2005. ACM.

[5] U. Bayazit. Macroblock data classification and nonlinear bit count estimation for low delay H.263 rate control. In *Proc. of IEEE International Conference on Image Processing (ICIP'99)*, pages 263–267, Kobe, Japan, October 1999.

[6] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *Proc. of NetGames'04*, pages 144–151, Portland, OR, August 2004.

[7] BerliOS. Freenx - nx components. http://openfacts2.berlios.de/wikien/index.php/Berlios Project:FreeNX - NX Components, 2008.

[8] A. Boukerche and R. W. N. Pazzi. Remote rendering and streaming of progressive panoramas for mobile devices. In *Proceedings of the 14th annual ACM international conference on Multimedia*, MULTIMEDIA '06, pages 691–694, New York, NY, USA, 2006. ACM.

[9] C.-F. Chang and S.-H. Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing*, PCM '02, pages 1105–1111, London, UK, 2002. Springer-Verlag.

[10] S. E. Chen. Quicktime vr: an image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 29–38, New York, NY, USA, 1995. ACM.

[11] S. E. Chen and L. Williams. View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 279–288, New York, NY, USA, 1993. ACM.

[12] M. Claypool and K. Claypool. Latency can kill: precision and deadline in online games. In *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 215–222, New York, NY, USA, 2010. ACM.

[13] D. R. Commander. Virtualgl: 3d without boundaries the virtualgl project. http://www.virtualgl.org/, 2007.

[14] B. C. Cumberland, G. Carius, and A. Muir. Microsoft windows nt server 4.0 terminal server edition technical reference. Microsoft Press, 1999.

[15] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester. A hybrid thin-client protocol for multimedia streaming and interactive gaming applications. In *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '06, pages 15:1–15:6, New York, NY, USA, 2006. ACM.

[16] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. RFC Editor, 1996.

[17] F. Duguet and G. Drettakis. Flexible point-based rendering on mobile devices. *IEEE Comput. Graph. Appl.*, 24:57–63, July 2004.

[18] P. Eisert and P. Fechteler. Low delay streaming of computer graphics. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 2704 –2707, oct. 2008.

[19] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer. Using latency to evaluate interactive system performance. In *OSDI*, pages 185–199, 1996.

[20] K. Engel, T. Ertl, P. Hastreiter, B. Tomandl, and K. Eberhardt. Combining local and remote visualization techniques for interactive volume rendering in medical applications. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 449–452, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[21] K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *IEEE Visualization*, 1999.

[22] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 160–170, New York, NY, USA, 1996. ACM.

[23] E. Games. Unreal tournament 3. www.unrealtournament.com/.

[24] J. Garrett-Glaser. x264: The best low-latency video streaming platform in the world. http://x264dev.multimedia.cx/archives/249, January 2010.

[25] F. Giesen, R. Schnabel, and R. Klein. Augmented compression for server-side rendering. In *Proc. of VMV'08*, pages 207–216, Konstanz, Germany, October 2008.

[26] H.-C. Hege, A. Merzky, and S. Zachow. Distributed visualization with opengl vizserver: Practical experiences. Technical Report 00-31, ZIB, Takustr.7, 14195 Berlin, 2000.

[27] G. Hesina and D. Schmalstieg. A network architecture for remote rendering. In *DIS-RT*, pages 88–91. IEEE Computer Society, 1998.

[28] HP. Remote graphics software. http://h20331.www2.hp.com/hpsub/cache/286504-0-0-225-121.html, 2006.

[29] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 43:1–43:10, New York, NY, USA, 2008. ACM.

[30] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras. Platform for distributed 3d gaming. *Int. J. Comput. Games Technol.*, 2009:1:1–1:15, January 2009.

[31] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. Protected interactive 3d graphics via remote rendering. *ACM Trans. Graph.*, 23:695–703, August 2004.

[32] S.-U. Kum and K. Mayer-Patel. Real-time multidepth stream compression. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1:128–150, May 2005.

[33] G. Kurillo, R. Vasudevan, E. Lobaton, and R. Bajcsy. A framework for collaborative real-time 3d teleimmersion in a geographically distributed environment. In *Multimedia, 2008. ISM 2008. Tenth IEEE International Symposium on*, pages 111 –118, dec. 2008.

[34] A. M. Lai and J. Nieh. On the performance of wide-area thin-client computing. *ACM Trans. Comput. Syst.*, 24:175–209, May 2006.

[35] F. Lamberti and A. Sanna. A streaming-based solution for remote visualization of 3D graphics on mobile devices. *IEEE Trans. Vis. Comput. Graph.*, 13(2):247–260, 2007.

[36] F. Lamberti, C. Zunino, A. Sanna, F. Antonino, and M. Maniezzo. An accelerated remote graphics architecture for pdas. In *Proceedings of the eighth international conference on 3D Web technology*, Web3D '03, pages 55–ff, New York, NY, USA, 2003. ACM.

[37] Y. Lee and B. Song. An intra-frame rate control algorithm for ultra low delay H.264/AVC coding. In *Proc. of ICASSP'08*, pages 1041–1044, Las Vegas, NV, March 2008.

[38] M. Levoy. Polygon-assisted jpeg and mpeg compression of synthetic images. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 21–28, New York, NY, USA, 1995. ACM.

[39] M. Levoy. The digital michelangelo project. In *3-D Digital Imaging and Modeling, 1999. Proceedings. Second International Conference on*, pages 2 –11, 1999.

[40] J.-M. Lien, G. Kurillo, and R. Bajcsy. Skeleton-based data compression for multi-camera tele-immersion system. In *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I*, ISVC'07, pages 714–723, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] Y. Liu, Z. Li, and Y. Soh. A novel rate control scheme for low delay video communication of H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(17):68–78, January 2007.

[42] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[43] W. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 1999.

[44] W. R. Mark and et al. Post-rendering 3D warping. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, 1997.

[45] J. Marquez, J. Domenech, J. Gil, and A. Pont. Exploring the benefits of caching and prefetching in the mobile web. In *Proc. of WCITD'08*, Pretoria, South Africa, October 2008.

[46] I. M. Martin. Adaptive rendering of 3d models over networks using multiple modalities. Technical report, IBM Research, 2000.

[47] I. M. Martin. Arte—an adpative rendering and transmission environment for 3D graphics. In *MULTIMEDIA '00*, pages 413–415, 2000.

[48] L. McMillan. *An Image-Based Approach to Three Dimensional Computer Graphics*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 1997.

[49] L. McMillan and G. Bishop. Plenoptic modeling: an image-based rendering system. In *SIGGRAPH '95*, pages 39–46, 1995.

[50] S. Milani and G. Calvagno. A cognitive approach for effective coding and transmission of 3d video. In *Proceedings of the international conference on Multimedia*, MM '10, pages 581–590, New York, NY, USA, 2010. ACM.

[51] H. Mobahi, S. R. Rao, and Y. Ma. Data-driven image completion by image patch subspaces. In *PCS'09: Proceedings of the 27th conference on Picture Coding Symposium*, pages 241–244, Piscataway, NJ, 2009.

[52] M. Nadeem, S. Wong, and G. Kuzmanov. An efficient realization of forward integer transform in H.264/AVC intra-frame encoder. In *Proc. of SAMOS'10*, pages 71–78, Samos, Greece, July 2010.

[53] I. Nave, H. David, A. Shani, Y. Tzruya, A. Laikari, P. Eisert, and P. Fechteler. Games@large graphics streaming architecture. In *Consumer Electronics, 2008. ISCE 2008. IEEE International Symposium on*, pages 1 –4, april 2008.

[54] Y. Noimark and D. Cohen-Or. Streaming scenes to mpeg-4 video-enabled devices. *IEEE Comput. Graph. Appl.*, 23:58–64, January 2003.

[55] Nvidia. Reality server. http://www.nvidia.com/object/realityserver.html.

[56] C. Ohazama. Opengl vizserver white paper. Silicon Graphics, Inc, 1999.

[57] OnLive. Onlive support documents. http://www.onlive.com/.

[58] S. Paik. Microsoft opengl information. http://www.opengl.org/resources/faq/technical/mslinks.htm.

[59] B. Paul, S. Ahern, W. Bethel, E. Brugger, R. Cook, J. Daniel, K. Lewis, J. Owen, and D. Southard. Chromium renderserver: Scalable and open remote rendering infrastructure. *IEEE Transactions on Visualization and Computer Graphics*, 14:627–639, 2008.

[60] S. K. Penta and P. Narayanan. Compression of multiple depth maps for ibr. *The Visual Computer*, 21:611–618, 2005. 10.1007/s00371-005-0337-8.

[61] S. G. Perlman, R. V. D. Laan, T. Cotter, S. Furman, R. McCool, and I. Buckley. System and method for multi-stream video compression using multiple encoding formats. US Patent No. 2010/0166068A1, July 2010.

[62] J. L. Phil Karlton, Paula Womack. Opengl graphics with the x window system (version 1.4). http://www.opengl.org/documentation/specs/, 2005.

[63] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England, and L. Nyland. The warpengine: an architecture for the post-polygonal age. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 433–442, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[64] S. Prohaska, A. Hutanu, R. Kahler, and H.-C. Hege. Interactive exploration of large remote micro-ct scans. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 345–352, Washington, DC, USA, 2004. IEEE Computer Society.

[65] H. Reddy and R. Chunduri. MPEG-4 low delay design for HDTV with multi-stream approach. Master's thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), 2006.

[66] A. Redert, M. de Beeck, C. Fehn, W. Ijsselsteijn, M. Pollefeys, L. Van Gool, E. Ofek, I. Sexton, and P. Surman. Advanced three-dimensional television system technologies. In *Proceedings of 3DPVT'02*, pages 313 – 319, 2002.

[67] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2:33–38, January 1998.

[68] O. Riva and J. Kangasharju. Challenges and lessons in developing middleware on smart phones. *IEEE Computer*, 41(10):77–85, October 2008.

[69] C. Salzmann. Real-time interaction over the internet. In *Ph.D. Dissertation*. Ecole Polytechnique Federale de Lausanne, 2005.

[70] R. W. Scheifler and J. Gettys. The x window system. *ACM Trans. Graph.*, 5:79–109, April 1986.

[71] D. Schmalstieg. *The Remote Rendering Pipeline - Managing Geometry and Bandwidth in Distributed Virtual Environments*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 1997.

[72] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of slim: a stateless, thin-client architecture. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 32–47, New York, NY, USA, 1999. ACM.

[73] B.-O. Schneider and I. M. Martin. An adaptive framework for 3d graphics over networks. *Computers & Graphics*, 23(6):867 – 874, 1999.

[74] R. Schreier, A. Rahman, G. Krishnamurthy, and A. Rothermel. Architecture analysis for low-delay video coding. In *Proc. of ICME'06*, pages 2053–2056, Toronto, Canada, July 2006.

[75] J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 231–242, New York, NY, USA, 1998. ACM.

[76] S. Shimizu, M. Kitahara, H. Kimata, K. Kamikura, and Y. Yashima. View scalable multiview video coding using 3-d warping with depth map. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(11):1485 –1495, nov. 2007.

[77] S. Singhal and M. Zyda. *Networked virtual environments: design and implementation*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[78] F. Smit, R. van Liere, S. Beck, and B. Froehlich. An image-warping architecture for vr: Low latency versus image quality. In *Virtual Reality Conference, 2009. IEEE*, pages 27 –34, march 2009.

[79] A. Smolic, K. Mueller, P. Merkle, C. Fehn, P. Kauff, P. Eisert, and T. Wiegand. 3d video and free viewpoint video - technologies, applications and mpeg standards. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 2161 –2164, july 2006.

[80] A. Snell and C. G. Willard. Ibm deep computing visualization. White Paper: http://www-06.ibm.com/systems/jp/deepcomputing/pdf/idc_white_paper.pdf.

[81] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. Widening the remote visualization bottleneck. In *Image and Signal Processing and Analysis, 2003. ISPA 2003. Proceedings of the 3rd International Symposium on*, volume 1, pages 174 – 179 Vol.1, sept. 2003.

[82] S. Stegmaier, M. Magallón, and T. Ertl. A generic solution for hardware-accelerated remote visualization. In *Proceedings of the symposium on Data Visualisation 2002*, VISSYM '02, pages 87–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[83] M. I. Technology. Thinanywhere. http:// www.thinanywhere.com, 2007.

[84] T. Tran, L. Liu, and P. Westerink. Low-delay MPEG-2 video coding. In *Proc. of VCIP'98*, pages 510–516, San Jose, CA, January 1998.

[85] S. University. Scanview : A system for remote visualization of scanned 3d models. http://graphics.stanford.edu/software/scanview/, 2007.

[86] G. K. Wallace. The jpeg still picture compression standard. *Commun. ACM*, 34(4):30–44, Apr. 1991.

[87] Y. Wang, J. Ostermann, and Y. Zhang. *Video Processing and Communications*. Prentice Hall, 1st edition, 2001.

[88] T. Wiegand, G. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[89] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[90] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The performance of remote display mechanisms for thin-client computing. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 131–146, Berkeley, CA, USA, 2002. USENIX Association.

[91] Z. Yang, Y. Cui, Z. Anwar, R. Bocchino, N. Kiyanclar, K. Nahrstedt, R. H. Campbell, and W. Yurcik. Real-time 3d video compression for tele-immersive environments. In *Proc. of SPIE/ACM Multimedia Computing and Networking (MMCN'06)*, 2006.

[92] Z. Yang, K. Nahrstedt, Y. Cui, B. Yu, J. Liang, S. hack Jung, and R. Bajscy. Teeve: the next generation architecture for tele-immersive environments. In *Multimedia, Seventh IEEE International Symposium on*, page 8 pp., dec. 2005.

[93] W. Yoo, S. Shi, W. Jeon, K. Nahrstedt, and R. Campbell. Real-time parallel remote rendering for mobile devices using graphics processing units. In *Proc. of ICME'10*, pages 902 –907, july 2010.

[94] I. Yoon and U. Neumann. Web-based remote rendering with ibrac (image-based rendering acceleration and compression). *Comput. Graph. Forum*, 19(3):321–330, 2000.

[95] M. Zhu, S. Mondet, G. Morin, W. T. Ooi, and W. Cheng. Towards peer-assisted rendering in networked virtual environments. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 183–192, New York, NY, USA, 2011. ACM.