Cloud-assisted Real-time Free Viewpoint Video Rendering and Streaming System

by

Yijian Wang

B.Eng., Nanjing University of Posts and Telecommunications, 2014

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

© Yijian Wang 2017

SIMON FRASER UNIVERSITY

Spring 2017

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced, without authorization, under the conditions for Fair Dealing. Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: **Yijian Wang**

Degree: **Master of Applied Science**

Title: ***Cloud-assisted Real-time Free Viewpoint Video Rendering and Streaming System***

Examining Committee: **Chair:** Paul Ho
Professor

Jie Liang
Senior Supervisor
Professor

Jiangchuan Liu
Supervisor
Professor

Ivan Bajic
Internal Examiner
Associate Professor
School of Engineering Science

Date Defended/Approved: March 27, 2017

Abstract

Free Viewpoint Video (FVV) is an emerging type of video which allows user to choose viewpoint freely in three-dimensional scenes. Depth-image-based Rendering (DIBR) is a common method to generate FVV using both texture and depth information. However, FVV rendering is more time-consuming than the original video since it has higher computational complexity. In order to make FVV rendering in real-time, a cloud-assisted system is proposed, which leverages cloud and distributed computing. In addition, we use multithread programming to take full advantage of cloud resources. As a result, by deploying our system on the WestGrid cluster, the FVV generation speed can be over 30 fps. Furthermore, to achieve the optimal trade-off between economic cost and user experience, we formulate and build mathematical models for the cloud-based FVV rendering and streaming system. Based on that, dynamic resource allocation algorithms are designed, which can provide the optimal resource allocation scheme according to users' requests. The performance of the system is demonstrated by various experiments. To the best of our knowledge, this is the first cloud-assisted real-time FVV rendering and streaming system.

Keywords: Free Viewpoint Video; cloud computing; multithread processing; dynamic resource allocation; optimization algorithm

Acknowledgements

First of all, I would like to thank my supervisor Dr. Jie Liang for his guidance and patience when I wrote my thesis in very limited time. He gave me a lot of inspirations and suggestions on this thesis during my master study.

I would also like to express my sincere thanks to Dr. Paul Ho, Dr. Jiangchuan Liu and Dr. Ivan Bajic for being my committee member and reviewing my thesis. Your suggestions are very precious and helpful to me.

I am also very grateful to my manager Sharon Om in SAP, who gave me much support and understanding to help me finish my thesis while working in SAP. I learned a lot from her on time management as well as coding skills.

Last but not least, I would like to acknowledge my friends in SFU and colleagues in SAP, who gave me additional help and advices to make my thesis and code better. I would also like to thank my parents who care about my life and health in China. I cannot make today’s achievements without their love.

Table of Contents

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Tables vii
List of Figures viii
List of Acronyms ix

Chapter 1. Introduction 1
1.1. Thesis Structure 3
1.2. Contributions 3

Chapter 2. Background 5
2.1. Free Viewpoint Video Rendering 5
2.2. Multimedia Cloud Computing 8
2.3. Multithread Processing 11
2.4. Summary 13

Chapter 3. Cloud-assisted Free Viewpoint Video Rendering and Streaming Achitecture 15
3.1. System Architecture 15
3.1.1. Video Capture 16
3.1.2. Cloud Processing 17
3.1.3. Client Interaction 20
3.2. Real-time solution 23
3.2.1. Dynamic Resource Allocation Scheme 23
3.2.2. Multithread Processing Strategy 28
3.3. Summary 30

Chapter 4. Mathmatical Modeling and Algorithm Design 31
4.1. Basic Assumptions and Strategy 31
4.2. Cost and User Experience Model 35
4.2.1. Cost Model 35
4.2.2. User Experience Model 36
4.2.3. Objective Function 39
4.3. Prediction Method and Resource allocation 42
4.3.1. Short-term Prediction and Resources Reallocation 42
4.3.2. Long-term Prediction and Resource Provision 44
4.4. Algorithm Design 46
4.5. Summary 50

Chapter 5. Implementation and Experimental Results 51
5.1. System Implementation 51
5.1.1. Free Viewpoint Video Synthesis 51
5.1.2. Video Coding 52
5.1.3. Multithreading Libraries 53
5.1.4. Video Streamer Development 54
5.1.5. Cloud Deployment 55
5.2. Experimental Results 56
5.2.1. Video Quality 56
5.2.2. Generation Speed 57
5.2.3. Users' Requests 59
5.2.4. Short-term Resources Reallocation 62
5.2.5. Long-term Resources Provision 64

Chapter 6. Conclusion and Future Work 68

References 69

List of Tables

Table 3.1 Example of Request Content 21

Table 4.1 Short-term resources reallocation algorithm (basic version) 47

Table 4.2 Short-term resources reallocation algorithm (with highlights solution) 48

Table 4.3 Long-term resources provision algorithm 49

Table 5.1 Relationship between bit rate and PSNR 56

Table 5.2 Relationship between threads and generation speed 58

Table 5.3 Relationship between bit rate and generation speed 59

Table 5.4 Testing parameters in Equation 4.27 63

Table 5.5 Initialized VMs among viewpoints 63

Table 5.6 Reallocated VMs among viewpoints 64

Table 5.7 Provisioned VMs among viewpoints 66

List of Figures

Figure 1.1. Fundamental concept of multimedia cloud computing [3] 2
Figure 2.1 A typical set up to capture Free Viewpoint Video [12] 6
Figure 2.2 Regular video frame and the according depth image 7
Figure 2.3 Cloud-based FVV rendering for mobile phone [20] 10
Figure 2.4 A process with two threads of execution, running on a single processor [25] 11
Figure 2.5 Memory space used by threads and processes [26] 12
Figure 2.6 Single process with single thread and single process with three threads [28] 13
Figure 3.1 Architecture of Cloud-assisted FVV Rendering and Streaming System 16
Figure 3.2 Example of Capture Cameras Alignment 17
Figure 3.3 Architecture of the Cloud-side Processing System 19
Figure 3.4 Interaction between Cloud-side and Client-side 22
Figure 3.5 Distribution of Users and Allocation of Provisioned Resources ... 24
Figure 3.6 Reallocation of Resources among Viewpoints 26
Figure 3.7 Dynamic Resource Allocation for the Highlight Viewpoint 27
Figure 3.8 Multithread Processing Workflow 29
Figure 4.1 Timeline of one renting cycle 33
Figure 4.2 Resources provision starting from the lower bound 46
Figure 4.3 Resources provision starting from the state of previous renting cycle 46
Figure 5.1 Bit rate and PSNR curve 57
Figure 5.2 Number of threads and generation speed curve 58
Figure 5.3 Distribution of users' requests among viewpoints 60
Figure 5.4 Users' requests within one renting cycle 61
Figure 5.5 Distribution of users' requests within one renting cycle 61
Figure 5.6 Distribution of users' requests with highlight in one renting cycle 62
Figure 5.7 Comparison of dynamic and static resource allocation 64
Figure 5.8 Total number of users' requests in two renting cycles 65
Figure 5.9 Distribution of users' requests with highlight in one renting cycles 65
Figure 5.10 Comparison of the optimal solution and $\boldsymbol{N \pm 1}$ cases 66

List of Acronyms

API	Application Programming Interface
ARIMA	Autoregressive integrated moving average
ARMA	Autoregressive moving average
DIBR	Depth-image-based Rendering
FPS	Frames Per Second
FVV	Free Viewpoint Video
GoP	Group of Pictures
MPEG	Moving Pictures Experts Group
MPI	Message Passing Interface
OpenCV	Open Source Computer Vision
OpenMP	Open Multi-Processing
Pthreads	POSIX Threads
VM	Virtual Machine
VSRS	View Synthesis Reference Software

Chapter 1.

Introduction

Since television was invented in the late 19th century and early 20th century, a great number of significant video technologies have been developed and applied to this area. From black-and-white to color, from mechanical to electronic and digital, television is giving viewers better contents and watching experience. However, these contents are still restricted to two-dimensional scenes and the viewers can only watch them from one single viewpoint. The way people enjoy television and visual media has not been changed in the last decades until the innovation of Free viewpoint television (FTV) and Free viewpoint video (FVV) technology.

Free viewpoint television is a system for viewing free viewpoint video, which allows the user to interactively control the viewpoint and watch virtual video contents from any 3D position. There is a very similar technology in computer-simulated video is known as virtual reality (VR) [**1**]. But there is still small difference between VR and FVV. With VR, the viewer can watch the whole surrounding 3D world from one point where the camera is set up. However, with FVV, the view can switch to different viewpoints to watch a same object from different positions and angles within the 3D scene. Hence, VR is mostly used in the immersion video games while FVV is widely applied in sports and concerts broadcasting.

As a new active area in computer graphics, FVV is drawing great attention from both users and researchers. It not only introduces unprecedented watching experience to users, but also makes researchers to face challenges from a whole new level. In order to generate FVV, multiple cameras need to be set in different viewpoints, and a particular algorithm and system is designed to synthesize the desired virtual view. The users apparently do not want much delay from this rendering process. Thus, generating FVV in

real-time or nearly real-time is our goal, which requires necessary hardware support to achieve. Considering different users may watch FVV on various devices with different hardware and most of them do not have enough computing power to render FVV in real-time, cloud computing comes to the rescue.

Cloud computing is an emerging computing technology based on Internet that provides various computing and storage service to computers and other devices on demand [**2**]. By using cloud computing, the users can take advantage of the resources in the cloud no matter what devices they are currently using. As result, even a mobile device user can obtain powerful CPU and GPU to watch FVV in real-time on their devices. However, the user or the FVV provider needs to pay for the cost to leverage such cloud resources. And in fact, better FVV watching experience usually requires more powerful and expensive hardware. Therefore, the trade-off between user experience and cost becomes one of our major concerns in this thesis.

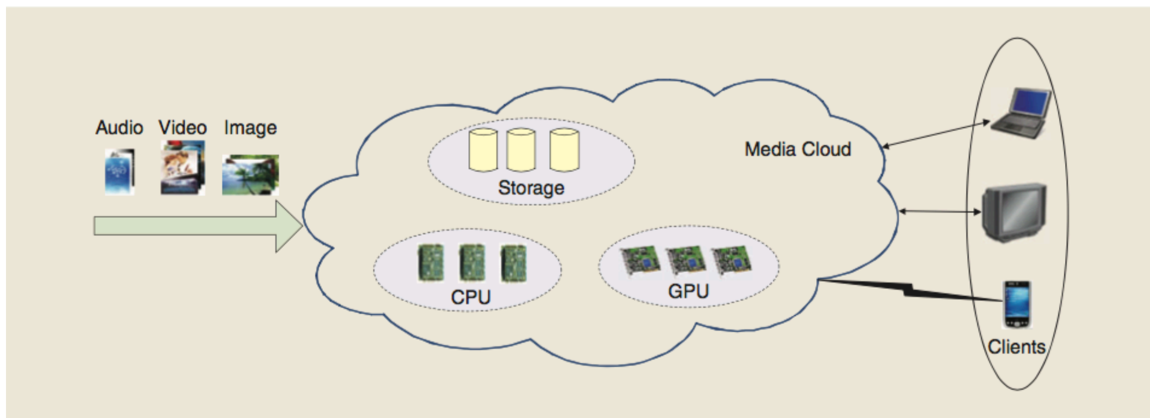


Figure 1.1. Fundamental concept of multimedia cloud computing [3]

To achieve the optimal trade-off, a FVV rendering and streaming system along with an algorithm is developed in this thesis. It is aimed to achieve the best Quality of Experience (QoE) by using as little resource as possible. In detail, this issue can be divided into two sub-problems:

1. How to reduce the amount of resources that need to be provisioned
2. How to fully utilize the resources we currently have

To solve these two problems, mathematical modeling, algorithm design and implementation techniques are involved and introduced in each separate section.

To integrate the real-time FVV rendering and cloud computing, there is an implementation technique called multithreading processing. With this technique, we can take full advantage of the hardware resources provided by cloud to speed up our FVV rendering process. Pthread, OpenMP, MPI are typical libraries to support multithreading programming. Through this way, we can develop a complete cloud-assisted real-time FVV rendering system, where hardware, software and algorithm work closely and efficiently.

1.1. Thesis Structure

Chapter 2 begins with the background and fundamental knowledge of FVV rendering and cloud computing. Then it explains how these techniques can be combined together to achieve our real-time FVV rendering system; Chapter 3 focuses on the architecture design of our cloud-assisted FVV rendering and streaming system. It presents each module with details and describes how we allocate cloud resources and divide the processing tasks; In Chapter 4, we formulate the problem and build mathematical models for precise analysis. Based on the objective function which quantifies the trade-off between economic cost and user experience, we propose an algorithm to figure out the optimal solution; In Chapter 5, the implementaion details of our cloud-assisted FVV rendering and streaming system are presented. Then we test our system in a practical environment with certain settings on parameters to verify its performance. Chapter 6 concludes this thesis and proposes serval ideas of future work.

1.2. Contributions

Firstly, we design a system based on the cloud and distributed computing that can dynamically allocate cloud resources to realize real-time FVV synthesizing and streaming;

Secondly, we develop a demo using C++ multithread programming and Qt based on the VSRS, OpenCV, FFmpeg, Pthread/OpenMP/MPI and NGINX, which can produce H.264 encoded FVV stream in real-time;

Thirdly, we configure and test the demo on the WestGrid cluster, where the generation speed can be over 30 fps. Also, the dynamic resource allocation algorithms in our system are verified to achieve the optimal trade-off between economic cost and user experience. To the best of our knowledge, this is the first cloud-assisted real-time FVV rendering and streaming system.

Chapter 2.

Background

In this chapter, we first introduce fundamental background of rendering free viewpoint video (FVV) and describe the technique that we use in this thesis. Then we give an overview of cloud computing and explain how it can be applied to multimedia processing, especially to the FVV rendering. After that, we demonstrate the basics in multithreading technology and describe how it can help with real-time tasks. Finally, we present our conclusion and the initial plan based on these backgrounds.

2.1. Free Viewpoint Video Rendering

Traditional video is recorded from a single fixed viewpoint, which confines the viewer to a flat two-dimensional image on the display. Free viewpoint video (FVV) breaks this restriction by providing three-dimensional watching experience. FVV is usually synthesised from a set of videos which are captured by multiple real cameras set in different positions. A typical set up is shown in Figure 2.1. With these original videos, we can synthesis FVV at virtual viewpoints using computer graphics approach [**4**]. Up to now, one of the most widely used methods is called Image-based Rendering (IBR).

FVV synthesized using IBR approach is aimed to solve the problem of generating novel viewpoints from a limited set of images taken from different positions. In such schemes, the cameras must be both calibrated and synchronised. In other words, the positions of the cameras in the scene and their direction and focal length are known, and each frame is captured at the same instant by each camera [**5**]. As a process of synthesizing novel views from camera images, IBR uses Light Field technique [**6**], which generates novel views by resampling camera images independent of scene geometry. Given enough images from different viewpoints, IBR can reconstruct arbitrary views in the scene. At the very beginning, researchers attempted to use large camera arrays and narrow baselines, and use interpolation or warping to obtain novel viewpoints. In [**7**], a dome of 51 cameras is first used to capture views from multiple positions. And one of the

first IBR system is designed in [**8**]. However, a major issue of IBR gradually shows up, that is it requires a large quantity of real cameras to record images from many different viewpoints in order to attain the rendering quality. Then the computer vision algorithm based on IBR was developed to mitigate this problem. Matusik et al. introduced model and image based rendering methods based on the visual hull [**9**]. Rendering approaches based on image depth maps [**10**] and 3D scene geometry model [**11**] can achieve high quality results with relatively small number of real cameras.

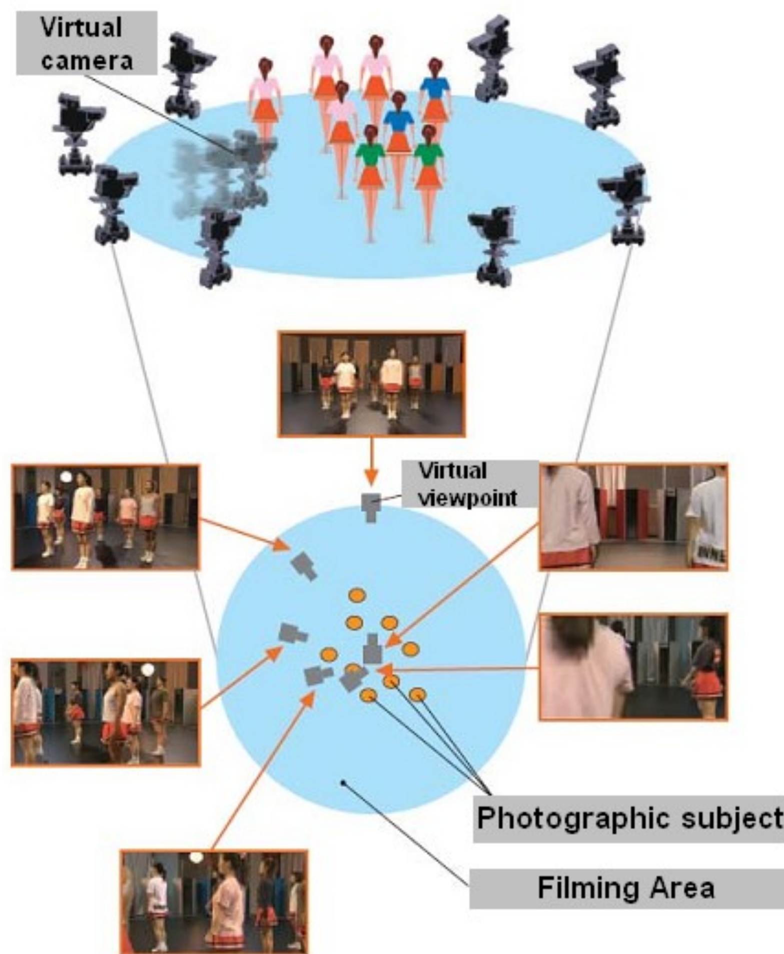
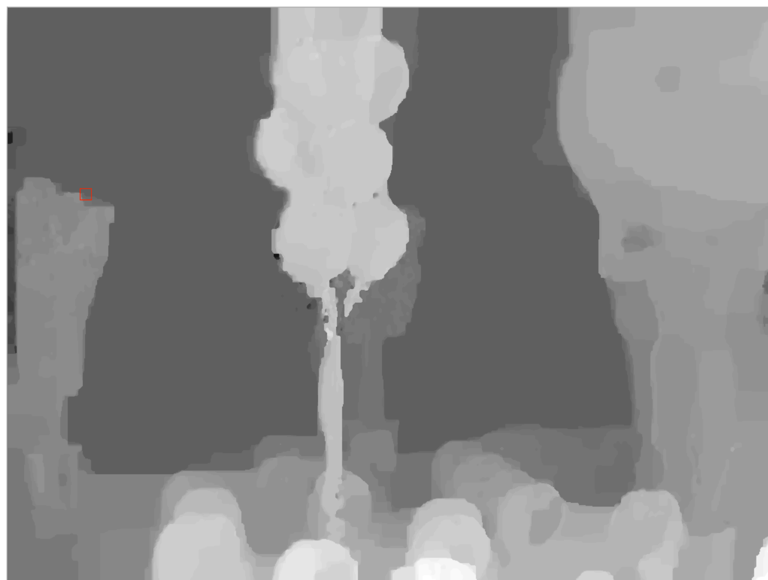


Figure 2.1 A typical set up to capture Free Viewpoint Video [12**]**

Depth-image-based Rendering (DIBR) is an evolution of regular IBR technology, which uses depth cameras in addition to ordinary cameras to capture raw videos. Different from usual RGB color values, the depth cameras can provide depth value for each pixel [**13**]. An example is shown in Figure 2.2, (a) is a texture image which is a video frame

captured by regular camera, (b) is the corresponding depth map captured by depth camera. With the depth information, we can generate a 3D model in order to render virtual views.

(a)



(b)

Figure 2.2 Regular video frame and the according depth image

The most common DIBR method consists of four steps [**14**]:

1. Warping left and right camera views to a virtual view using the depth images;
2. Processing the projected images to resolve artifacts like ghost contours;
3. Blending the processed images. For each pixel in the virtual view, it should be generated from either left or right projected textures;
4. Inpainting the disocclusions in the resulting image.

Through these steps, DIBR can generate high quality virtual view using the texture and depth images captured by relatively small number of cameras. So, we select it as the FVV rendering approach in this thesis. However, the speed of IBR and DIBR is highly depends on the computational power. Image processing is performed on both texture and depth images. The traditional rendering system can hardly produce FVV frames in real-time [**15**]. To speed up this processing and make it real-time or close to real-time, we need to take advantage of powerful hardware resources. Using cloud computing technique is one feasible solution to this issue. In this thesis, we integrate it with DIBR to build our cloud-assisted FVV rendering system.

2.2. Multimedia Cloud Computing

Cloud Computing is a type of parallel and distributed computing technology. A cloud usually consists of a collection of inter-connected Virtual Machines (VM). These VMs are dynamically provisioned and provided by the service provider as computing resources to consumers via the Internet. A combination of VMs makes up a cluster or a grid, then forms a cloud. Nowadays, more and more market-oriented Clouds are appearing, including Amazon Elastic Compute Cloud (EC2), Google App Engine, Microsoft Live Mesh. Through these platforms, the customers can pay to rent cloud resources when they need, without building and maintaining complex hardware infrastructure [**16**].

With the development of Web 2.0, multimedia applications and services are all over the Internet and mobile wireless networks. The high demand of multimedia requires significant computational power to process the data and stream to the clients. In the emerging cloud-based multimedia computing framework, the users can store and process their multimedia data in the cloud in a distributed manner. Generally, a media cloud is

equipped with a large number of high performance hard disks, CPUs and GPUs. Under the control of the resource allocator and load balancer, these resources are dynamically provided to the Media Service Providers (MSP). The MSPs can leverage the cloud to serve users with online storage, multimedia streaming, etc. For the mobile users, this media cloud can dramatically alleviate the burden of computation on the client-side and help with saving the battery of their devices [**3**]. With cloud computing, some high-computational-cost multimedia applications and services like FVV can be even completely conducted in the cloud, which breaks the traditional hardware limitations. Even mobile users without strong CPU and GPU support can watch FVV on their devices.

With cloud computing, we can also speed up the usual image and video processing. The major idea is to distribute the processing tasks in a cluster for parallel computing. However, it is not a new concept. A MapReduce based data processing paradigm on large clusters is introduced in [**17**], which uses a Map and Reduce function to automatically parallelize the computation across large-scale clusters of machines. In [**18**], a Split and Merge architecture is developed to perform video encoding on cloud. The basic idea is to split the original video file into several chunks based on key frames. Then it perfroms encoding on each chunk in parallel to reduce the time cost. Finally all encoded chunks are merged into one output video file.

As a particular type of video, free viewpoint video can also leverage the cloud computing technology to process frames in parallel in the cluster to reduce the time of rendering. In this thesis, we will demonstrate how we implement a FVV rendering application on a cloud cluster in the following chapters. Besides the time consumption, Quality of Service (QoS) is another important issue in cloud computing. Since a cloud can be used by various users to run a large number tasks at the same time, we need an efficient way to provision the resources and schedule the tasks. Otherwise, some of the users may experience long response time, which causes relatively low QoS. To solve this issue, the queuing model has been widely used. In [**19**], an approach to optimize the resource allocation on multimedia cloud is developed based on queuing model. It models the service process at multimedia cloud data center as three queuing systems and uses mathmatical way to get the optimized solution to minimize the mean response time and the resource cost. For FVV rendering, a framework for mobile devices is introduced in

[**20**]. Figure 2.3 illustrates its concept. Each time when user send out a view request, the cloud will render the view and sent the stream back to user. To minimize the interaction delay, the client will perfrom local rendering as well. Then the authors used convex optimization to determine the optimal job balance betwwn cloud and client. In addition, the concept of cloud-assisted view synthesis was introduced in [**21**] and [**22**], but neither of them implemented in the system.

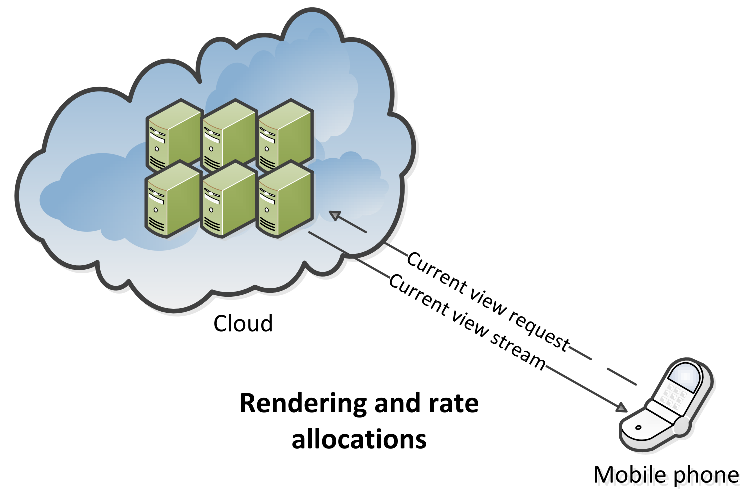


Figure 2.3 Cloud-based FVV rendering for mobile phone [20**]**

The ultimate goal of reducing the response time is to make it real-time, which is possible to achieve by leveraging the power of cloud computing. Theoretically, we can split the task to multiple small enough sub-tasks, which can be finished in real-time. However, this method may need many resources in cloud and the cost would be relatively high. Focusing on real-time tasks, resources with different speeds and costs are addressed in [**23**] and an optimal solution is provided to minimize the economic cost and meet all the deallines of the tasks. The introduction of multimedia cloud gives us an overview of cloud computing and how it can be applied to FVV rendering. But there is still a detailed technique in the implementation of speeding up a task using cloud, that is multithread processing.

2.3. Multithread Processing

In computer science, a thread is the smallest unit of sequence of programmed instructions that can be managed independently by the operating system [**24**]. There is another similar concept called process. A process is also an unit of program which can be performed by system individually. But in fact, these two concepts are different in many aspects. Generally, a thread is a component of a process in most cases, which is depicted in Figure 2.5.

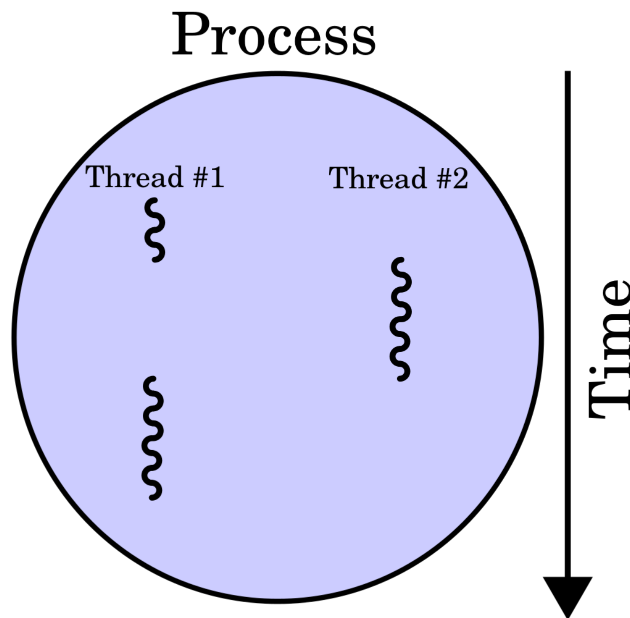


Figure 2.4 A process with two threads of execution, running on a single processor [25**]**

A process may contain multiple threads which can be executed concurrently, separately and mutually exclusively in time. These threads have shared resources including memory, while different processes use separate resources in the system, which is shown in Figure 2.6. A thread is also called as a lightweight process. It provides a way to improve program running performance through parallel computing. This approach is named as multithread processing.

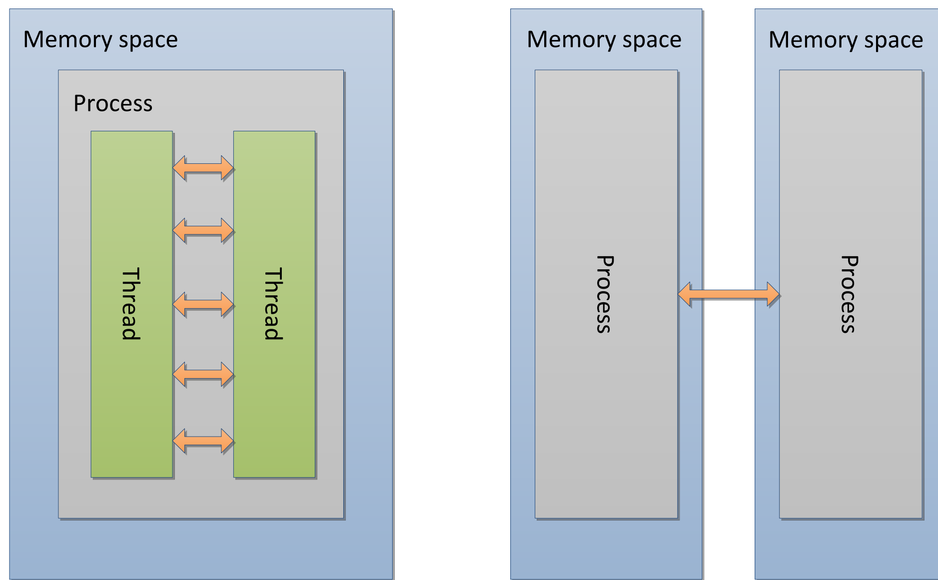


Figure 2.5 Memory space used by threads and processes [26]

In computer programming, single threading is the processing of one command at a time. The opposite of single threading is multithreading. It allows a process to have multiple threads which can be run independently. Systems with only a single processor generally implement multithreading by time slicing. The multithreads can be switched in the processor very quickly, which makes it looks like all threads are running concurrently. Nowadays, with the rapid growth of hardware technology, more and more computers are equipped with a multi-core processor or multiple processors, which means we have multiple processor units to run multithreads program. With this kind of processor, each thread can be executed on each core of the processor. As result, a multithread task can be performed in parallel to reduce the running time. In addition, now most of processors have ability of hyper-threading, which is technology to make a single physical CPU perform as two logic CPUs to the operating system [**27**].

With this hardware support, the program can be faster and more efficient. For example, considering two scenarios illustrated in Figure 2.7, the same single process can be run with single thread on one processor while it can also be run with three threads on three processors. Since multithreads can share most resources in the process, each one of them can access and process part of the task and reduce the time cost to one third of the original. Although multithreading may introduce more initialization and resource

allocation time as each thread need to be assigned with register, counter and stack separately, it still can save a lot of time when running a large number of tasks. What we need to do is migrate the programs into multithreading way and compile them with hardware support. This concept perfectly matches with multimedia cloud computing, since by leveraging cloud we can use multiple processors for our tasks. Then we can take advantage of them to make our multithreading program. Finally, with cloud computing and multithreading technique, our tasks can be run much faster, even close to real-time.

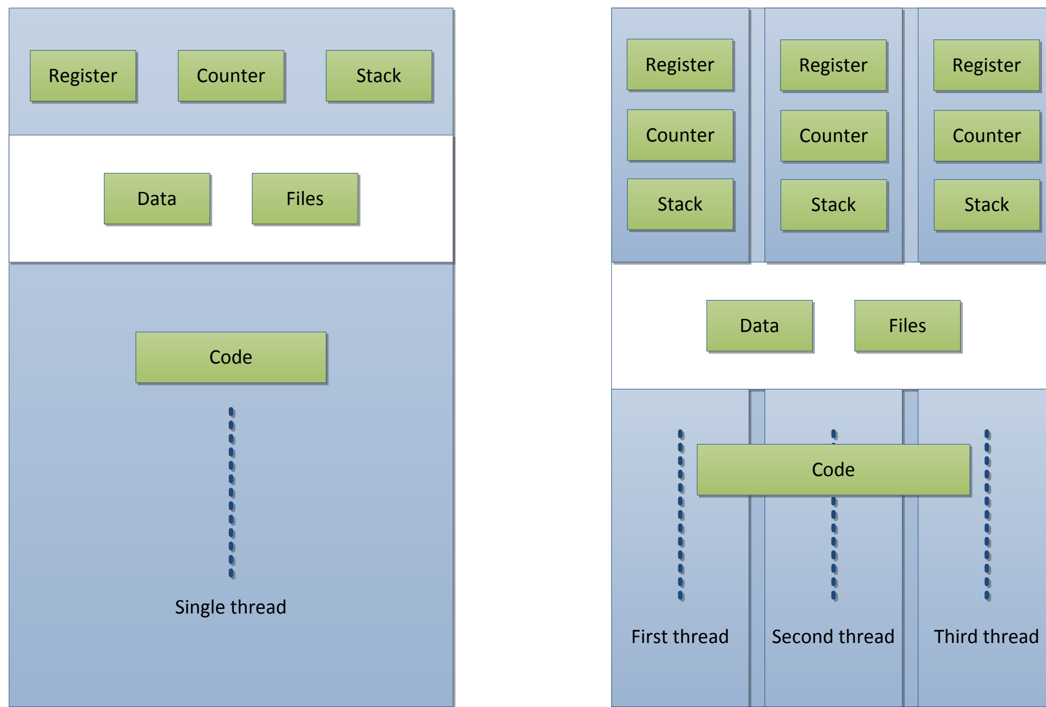


Figure 2.6 Single process with single thread and single process with three threads [28]

2.4. Summary

The current DIBR system has the ability to generate good quality FVV using relatively few cameras. Nevertheless, the processing speed highly relies on the hardware and it can hardly produce FVV in real-time. With the development of Internet, cloud computing becomes an important and reliable approach for specific users to resolve hardware limitations. The users can leverage the massive resources in a cloud to support their local tasks. However, using cloud resources is not free. To reduce the economic cost,

it is necessary to design an efficient and smart system to allocate the resources and schedule the tasks. Meanwhile, the FVV rendering program should take full advantage of the recourses provisioned from cloud. Multithread processing is an essential technique. Since nowadays most of computers and virtual machines are equipped with multi-core processors and multiple virtual CPUs, there is a big potential in the hardware. To release this power, we need to make our current software to be multithreading. Theoretically, the running time can be reduced by multiple times if the program can be run on multiple threads. Hence, it is a feasible method to make current FVV rendering system to perform in real-time on cloud.

To achieve this goal, we need to perform the following steps:

1. Optimizing the current DIBR software and changing it from traditional single thread processing to multithreads processing;
2. Developing a new multithreading FVV rendering system and migrating it from local to the cloud;
3. Designing a resource allocation algorithm for this system to minimize the economic cost but still maintain a high QoE and finish the tasks in real-time;
4. Implementing the whole system in code based on the support of certain open source libraries and tools.

This thesis will cover these steps in the next several chapters and present the experimental results at the end. To the best of our knowledge, there has never been a completely built real-time FVV rendering system which is assisted by cloud computing. Although some of the concepts and the technology used in this thesis may not be very novel, it is still a pioneering attempt to integrate them together. This is exactly the motivation to produce this thesis.

Chapter 3.

Cloud-assisted Free Viewpoint Video Rendering and Streaming Achitecture

In this chapter, we first introduce the whole architecture of our cloud-assisted FVV rendering and streaming system. Then the three major parts which are video capture, cloud processing and client interaction are described with details. After that, we demonstrate our solution to render FVV in real-time on the cloud. It covers the allocation scheme of cloud resources from the macroscopic and the division of processing tasks on multiple processors from the microscopic. Finally, we conclude and present the summary of this chapter.

3.1. System Architecture

The architecture of our cloud-assisted FVV rendering and streaming system is depicted in Figure 3.1. It consists of three major components, which are the capture side, cloud side and client side. Usually the raw texture views and depth maps are captured by multiple cameras set in the view scene. After that, these raw data are compressed and encoded before sending to the multimedia cloud through the Internet. The cloud has two kinds of core recourses. One is the storage resources, which are used for saving the original received data and maintaining the generated FVV data. The other one is computational resources, which are applied to processing the received data. In general, the cloud side first decodes the received data. Then the novel virtual viewpoint videos are synthesized based on the texture views and depth maps of neighboring views. Afterwards, these virtual videos are compressed and encoded again in order to send to the clients. At client side, there are possibly many different devices, such as desktop, laptop, TV and mobile devices. These clients can send out the request of specific viewpoints to the cloud. Then the cloud will send back the generated video steams to them. Here we should note that the request and response are asynchronous, since the requests received by cloud will be dealt with based on the queue model and it may take some time to generate the FVV on demand. In fact, there are some cases that the cloud will perform synthesis on

certain viewpoints even without any request and the output FVV will not be delivered to clients immediately. For example, when the cloud has some idle recourses, it will perform synthesis in advance on certain viewpoints and the generated video will be kept in cloud for a period of time to prepare for the delivery. When clients request for these viewpoints, the videos will be sent without the generating delay. Through this strategy, we can take the full advantage of cloud resources and achieve high Quality of Experience.

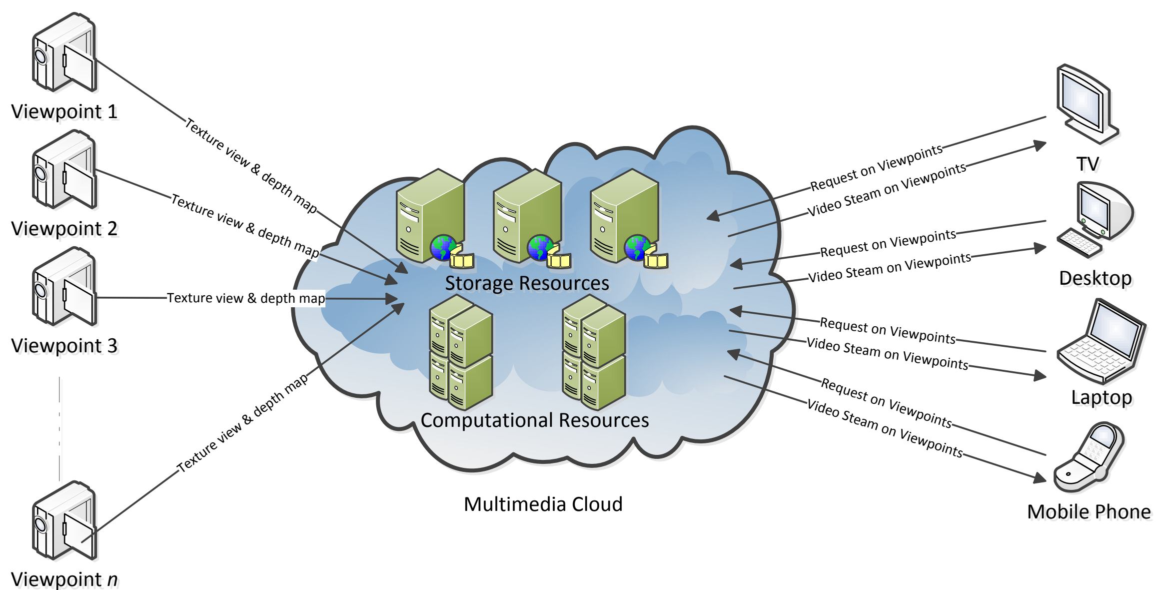


Figure 3.1 Architecture of Cloud-assisted FVV Rendering and Streaming System

3.1.1. Video Capture

The setup of capture cameras depends on the application scenarios. They can be aligned or circled for applications, such as concert, football game, etc. To achieve high quality FVV, there can be a large number of cameras which are not only set up in the two-dimensional space, but also the three-dimensional space. These cameras should be close to each other to avoid potential defects. Theoretically, two cameras are enough to generate a novel virtual view which is usually at the midpoint between them. Figure 3.2 illustrates a simple alignment of capture cameras. Suppose these cameras are set on a

line which represent viewpoints 1 to *n*, the virtual viewpoints can be between each two of them. Every adjacent two cameras provide the left view and the right view in order to synthesize the virtual view. This synthesis is based on the texture view and depth map captured by the left and right cameras. An additional set of parameters which reflects the settings of cameras is also required by this synthesis. This alignment model is the basic cameras setup in the following chapters as well.

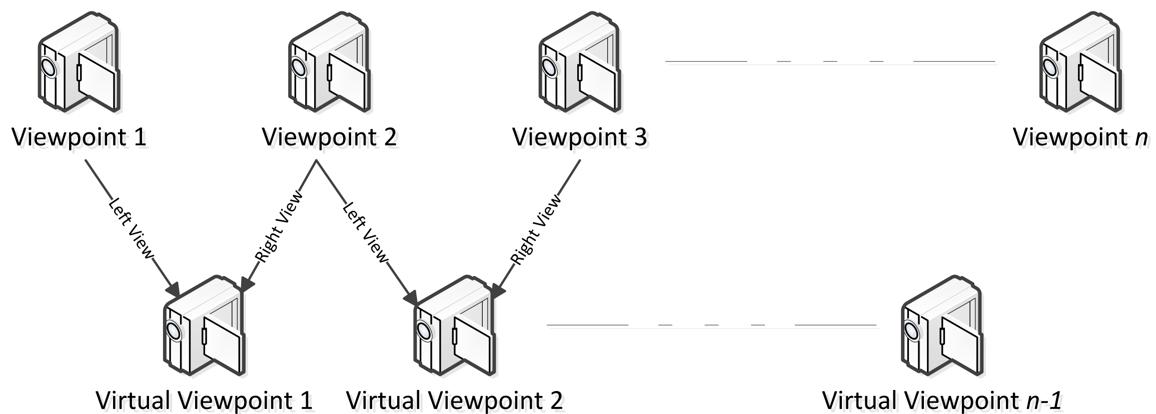


Figure 3.2 Example of Capture Cameras Alignment

To send the captured video to the cloud immediately, especially for the live video capturing and streaming, we can leverage certain real-time streaming protocol, such as Real-Time Messaging Protocol (RTMP). It allows the capture-side to deliver the videos with low latency as live video stream to the cloud-side.

3.1.2. Cloud Processing

Cloud processing is a core part in our cloud-assisted FVV rendering and streaming system. It receives captured video from the capture camera side and simultaneously outputs FVV stream based on the user request from the client side. This module does not only contain data processing, but also the data analysis and decision making. The whole architecture is shown in Figure 3.3.

As demonstrated in section 3.1.1, the original viewpoint videos are sent to the cloud after encoding and compression through the Internet. When these videos are

received by the cloud, it first uses the corresponding decoder to obtain the video data which includes both texture views and depth maps. Then the raw data is forwarded to the FVV synthesis module, which can generate novel virtual viewpoint using the captured texture and depth information. However, the output FVV is not applicable to storage and streaming since it is not compressed and does not have suitable video container to match the client-side devices. Hence, we encode it using specific codec in order to stream to the users. As described above, sometimes the generated FVV will not be steamed immediately, so it will be saved in the cloud-side storage and prepare for steaming on demand. The video streamer will handle the video container, since there are different requirements on container for various client-side devices. Based on the need, the encoded FVV will be encapsulated into corresponding container and steamed to the user. In addition, we should note that there is a multithreading coordinator working closely with this FVV generating process in order to speed it up to real-time. It receives the information including the provision of hardware resources from resources allocator, such as the number and the specification of vCPUs. This information determines how the multithreading can be coordinated, including multithread decoding and encoding, multithread FVV synthesis. It can also respond with the time cost of processing to the resources allocator in order to adjust and improve the overall performance.

To timely provide the multithreading coordinator with correct information, there are several other controlling modules which are resources allocator, task manager and predictive analyzer. These modules are working together for the decision making and data analysis. Here the core module is the task manager which is the entry point of user request from client-side. It gathers the video steam request on specific viewpoints every certain period of time. On one hand, the task manager is provided with the analytic results on the historical data, which is very useful to determine if it should add or reduce the cloud recourses to match the trend of the quantity of user requests in the future. On the other hand, it stores the collected requests into the predictive analyzer for the data analysis.

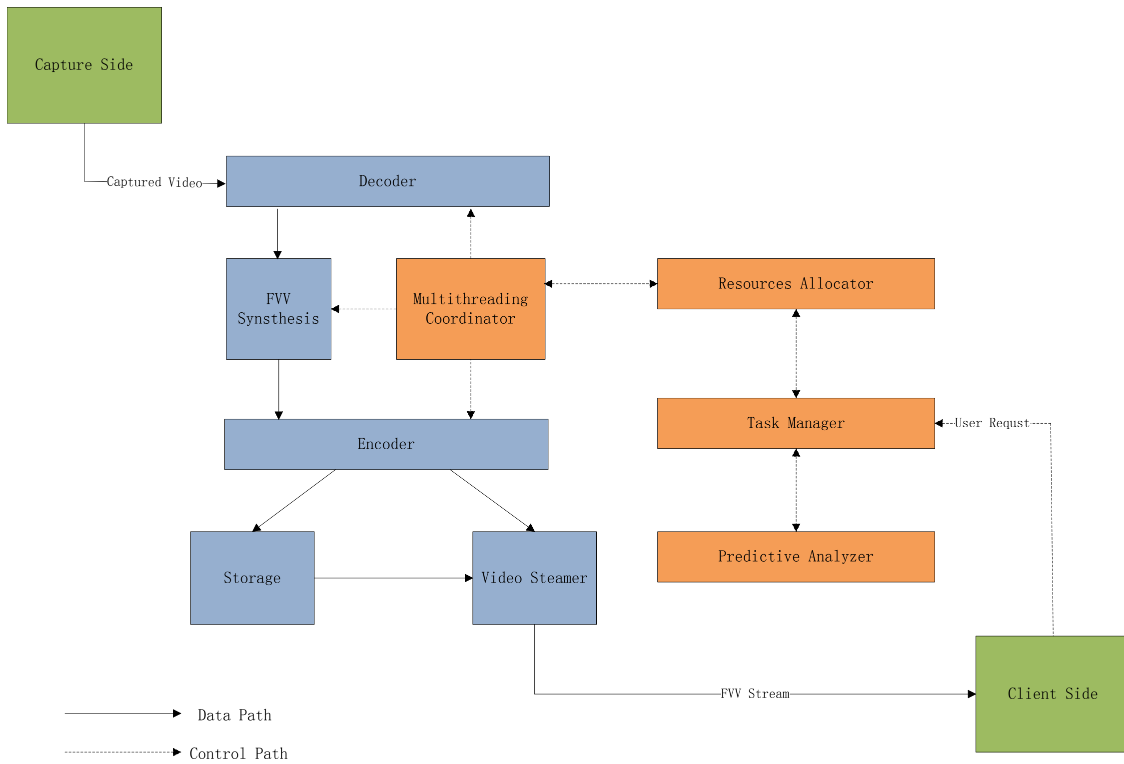


Figure 3.3 Architecture of the Cloud-side Processing System

The predictive analyzer basically performs two different kinds of prediction. The first one is short-term prediction which reflects the immediate fluctuation in user requests. It can be used to arrange the tasks among cloud recourses in the next serval time periods. However, in some situations, the user requests fluctuate frequently in short period of time, the prediction result is not reliable if we only depend on the short-term prediction. Also, considering adding or reducing cloud resources cannot be taken into effect immediately and renting resources usually charges on a long time [**29**], we introduce long-term prediction along with the short-term prediction. It relies on the historical data in a long time and can offer more convincing prediction on long-term trend. Based on the long-term prediction, we can determine to add or reduce cloud resources in the next renting period, which helps a lot to save cost and improve the user experience.

All the data analysis results including the short-term and long-term prediction are sent to the task and forwarded to resources allocator. Based on the results, the resources allocator can modify the resources arrangement among different tasks in the short period

of time. It can also change the recourse renting scheme in a long time. All these status and provision information will be provided to the multithreading coordinator so that it can arrange proper hardware for every FVV synthesis process. The multithreading coordinator responses with the performance results in return to the resource allocator. These results will also be forwarded to other control modules to make sure the processing is real-time.

Thanks to the shared memory and message passing inside the cloud cluster, these modules in the cloud-side can work closely to each other. To take full advantage of the cloud resources and minimize the cost and time delay, we need an efficient algorithm and prediction methods to be applied in the control modules. Also, the implementation of the cloud-side FVV rendering is another key to maintain the whole process real-time. All the details of algorithms, methods and implementation techniques will be introduced in the following chapters.

3.1.3. Client Interaction

On the client side, the user can send out FVV requests on viewpoints to the cloud and receive the generated FVV from it. Usually, the request contains at least viewpoint information that indicates which view stream should be synthesised. In addition, based on various client devices and network conditions, it allows user to select the video resolution and quality. In general, the FVV with high resolution and quality will take longer time for the processing and need higher speed network transmission. For the non-live video, the user may also select the specific sequences to watch within a FVV. Then the cloud does not need to process on the whole FVV, but only parts of it. Furthermore, the specifications of users' devices are sent along with the request. It includes information on the video codec and container which are supported by the client-side devices, which can help the cloud-side to choose proper video processing format. For the request transmission format, JavaScript Object Notation (JSON) and Extensible Markup Language (XML) are widely used nowadays. All the request information can be wrapped into these formats in order to be transmitted to the cloud. An example of request information is given in table 3.1.

On most of the current video websites, user can watch video streams on the website and web-based applications. Both these websites and applications are in the

front-end, they will gather users' requests and send them to the back-end. The communication between font-end and back-end is based on network protocol. For example, Hypertext Transfer Protocol (HTTP) and Real-Time Messaging Protocol (RTMP) are two famous protocols which are used for live video steaming. Apple implemented HTTP-based media streaming communications protocol called HTTP Live Steaming (HLS) for their products, including QuickTime, Safari, OS X, and iOS. It resembles MPEG-DASH and divides the overall stream into a sequence of small HTTP-based downloadable video files. Each of these video files contains short chunk of video stream which is in the format of MPEG transport stream (MPEG-TS). While they are played seamlessly, the users will feel like watching live video stream [**30**]. The RTMP is another option for the live video streaming which is owned by Adobe. It is based on Transmission Control Protocol (TCP) and can maintain persistent connections and low-latency communication. Flash Video (FLV) is the required video container while Advanced Audio Coding (AAC) and H.264/MPEG-4 AVC can be the corresponding audio and video encoding format [**31**].

Table 3.1 Example of Request Content

Objects in Request	Example Values
Request Viewpoint	Viewpoint 2
Video Quality	720p, 30fps
Video Sequence	From time 1:15
Video Codec	H.264
Audio Codec	AAC
Container	FLV

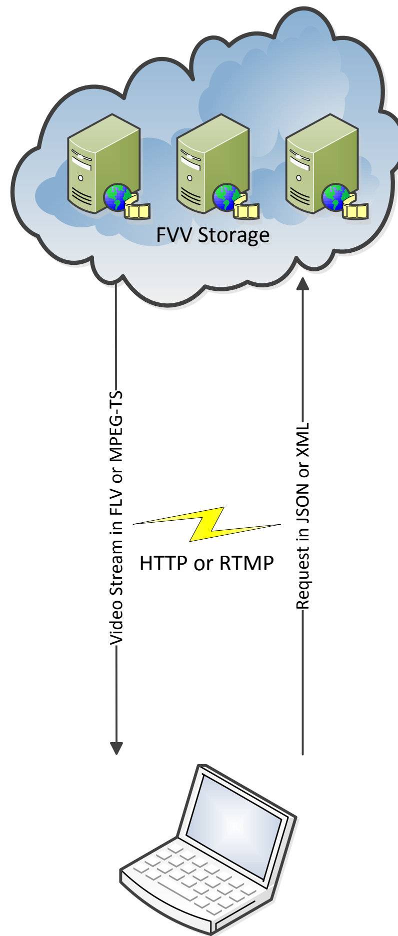


Figure 3.4 Interaction between Cloud-side and Client-side

The overall interaction between client-side and cloud-side is depicted in Figure 3.4. The users' requests are formatted in JSON or XML and sent to the cloud. As described above, the FVVs are saved in the storage resources in the cloud. When they received users' request, the corresponding FVV sequences will be delivered to the streamer and sent back to users as response. These sequences are formatted as users request and transmitted through the network with HTTP or RTMP. After retrieving the FVVs from the cloud, the client-side only needs to decode them and can watch them as normal video streams.

3.2. Real-time solution

Real-time FVV rendering and streaming is another key point in our system design. We introduce cloud computing and multithread computing to FVV generation in order to make it real-time. Based on the architecture proposed in the previous section, there are several modules taking charge of resource allocation and tasks management. In this section, we give detailed scheme and strategy to describe the logic which is used in these modules. Firstly, we describe how the resources are allocated among different viewpoints rendering. Secondly, we explain how these resources are applied to multithreading FVV generation. The ultimate goal is to make the whole FVV rendering process be able to complete in real-time.

3.2.1. Dynamic Resource Allocation Scheme

According to different cloud service providers, the cloud resources can be rented in different schemes. The most common one is to rent the resources in advance as provision. The quantity of provisioned resources depends on the number of viewpoints and the users, since every unit of cloud resources has limited processors, storage space and streaming bandwidth. Generally, these resources are represented as Virtual Machines (VM) in the cloud. Since cloud is a distributed system with resources at different locations. To reduce the delay, we want to use the VMs that close to the users. In addition, each VM has limited throughput which determines the number of requests it can handle simultaneously. Therefore, generally a viewpoint with more users will need more resources in the entire cloud system. On the client-side, the distribution of users' requests is usually not even among different viewpoints. For example, supposing a concert or a football game is streamed using FVV, most users will choose to watch through the viewpoints in the center, that is why the ticket price in these areas are often higher than the others. Based on the distribution of users, the cloud recourse should be provisioned specifically. In this thesis, we ignore the actual locations of users and group all VMs together. The number of VMs to provision is our focus.

In the ideal situation, all the viewpoints can be assigned with enough VMs to render and stream FVV. So, we can refer to the prediction results from the historical data to

determine the quantity of VMs needed as provision. The allocation of resources may be similar to Figure 3.5. Here we only consider the virtual viewpoints which need to be synthesized in the cloud. In Figure 3.5, we assume that the users are Gaussian distributed among the six example virtual viewpoints. Correspondingly, the provisioned cloud resources are distributed same as the users in order to achieve the optimal performance and generate FVV in real-time. By this allocation, the users can be served by the VMs effectively if the users distribution remains unchanged.

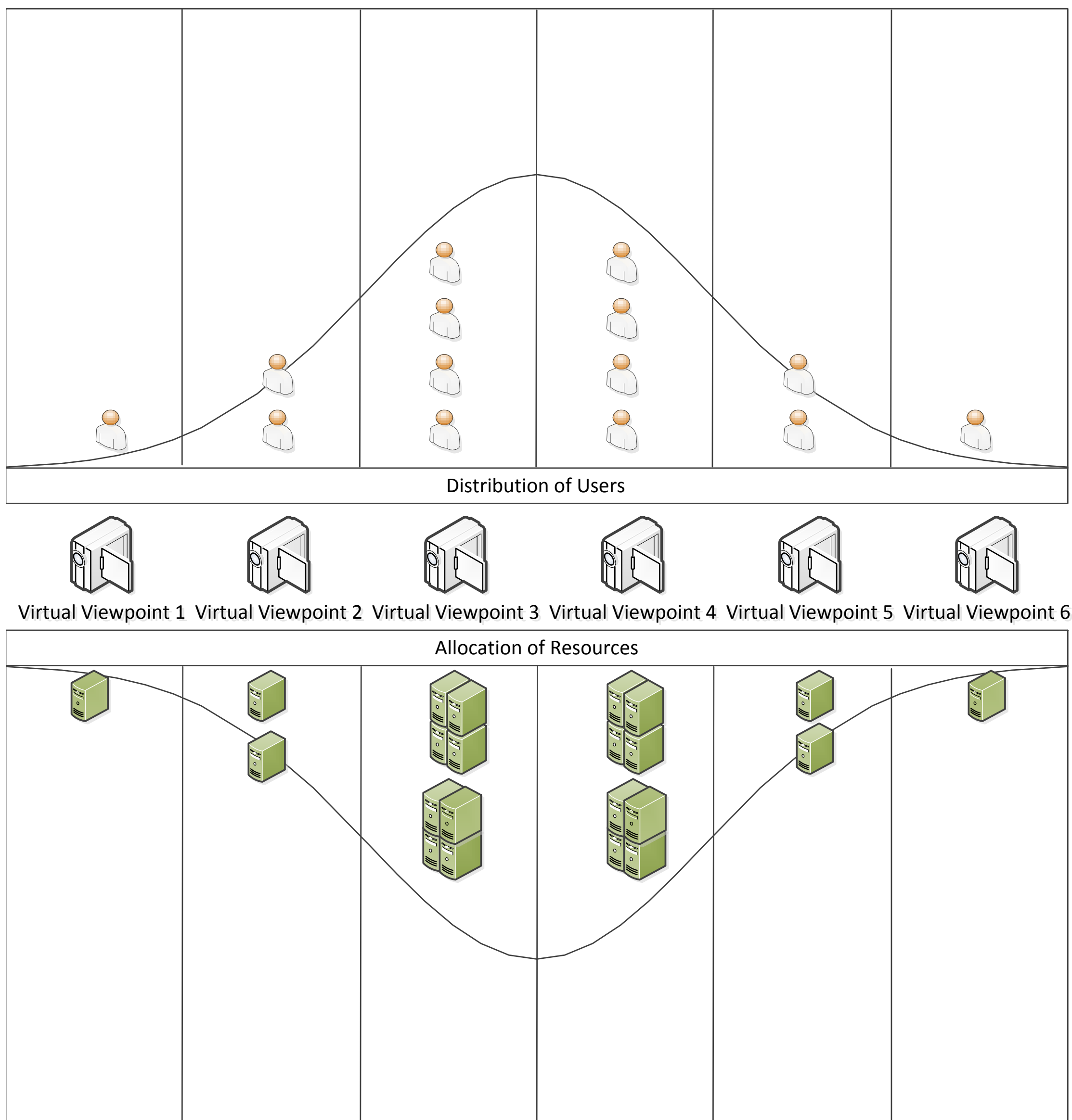


Figure 3.5 Distribution of Users and Allocation of Provisioned Resources

However, the allocation in Figure 3.5 can hardly remain as the optimal solution throughout the entire FVV streaming process. Firstly, the distribution of users usually keeps changing when the focus of the scenario varies. For example, when there is particular highlight happening in a football game, like penalty or goal, most of users will switch to the viewpoints close to there. It may cause the VMs serving that parts of viewpoints overloaded in a short period of time and the real-time streaming is stuck for a while. Secondly, even though the static resource allocation scheme can achieve the best rendering and streaming performance, it may not be the optimal real-time solution when we take economic cost into consideration. To reduce the cost, we cannot guarantee all the viewpoints have enough provisioned cloud resources at the beginning. But we can re-allocate them dynamically later if some of them finish their tasks before deadline. So here in this thesis, we propose a dynamic resource allocation scheme.

The first motivation to design our dynamic resource allocation scheme is to reduce the economic cost. In fact, we do not need to provision every viewpoint with sufficient VMs at the beginning, which means we can allocate less VMs for the viewpoints that are not popular as expected. For those viewpoints lack of provisioned VMs, our resource allocator can switch VMs from other viewpoints to them, if they can finish their tasks before the deadline. Before this reallocation happens, the users watching the viewpoints without enough VMs may experience longer delay and queuing time. And the length of this delay depends on how long the VMs on other viewpoints can finish the assigned tasks. This dynamic resources reallocation scheme can be illustrated in Figure 3.6. For example, assuming the virtual viewpoint 1 and 6 do not have provisioned VMs at the beginning of FVV streaming and the VMs provisioned for virtual viewpoint 3 and 4 are sufficient to finish tasks earlier than deadline, so we can reallocate them to other viewpoints, such as the virtual viewpoint 1 and 6. The Figure 3.6 is just an example, since actually all the VMs in virtual viewpoint 3 and 4 can be switched to other different viewpoints if their tasks are done, not only one part of them. Through this scheme, we can reduce the economic cost even though some of the users may experience streaming delay. However, we consider this delay as acceptable result, since it only happens on the least popular viewpoints, like the viewpoint 1 and 6 in Figure 3.6, which can minimize the defects on user experience. In addition, the overall optimal solution not only considers the user experience, but also the economic cost. There is a trade-off between these two. The optimal solution should

minimize the cost and maintain the overall user experience as high as possible at the same time. And the overall user experience mostly relies on the central viewpoints. We will give the mathematical model on this issue in the next chapter in order to demonstrate the optimal solution more precisely.

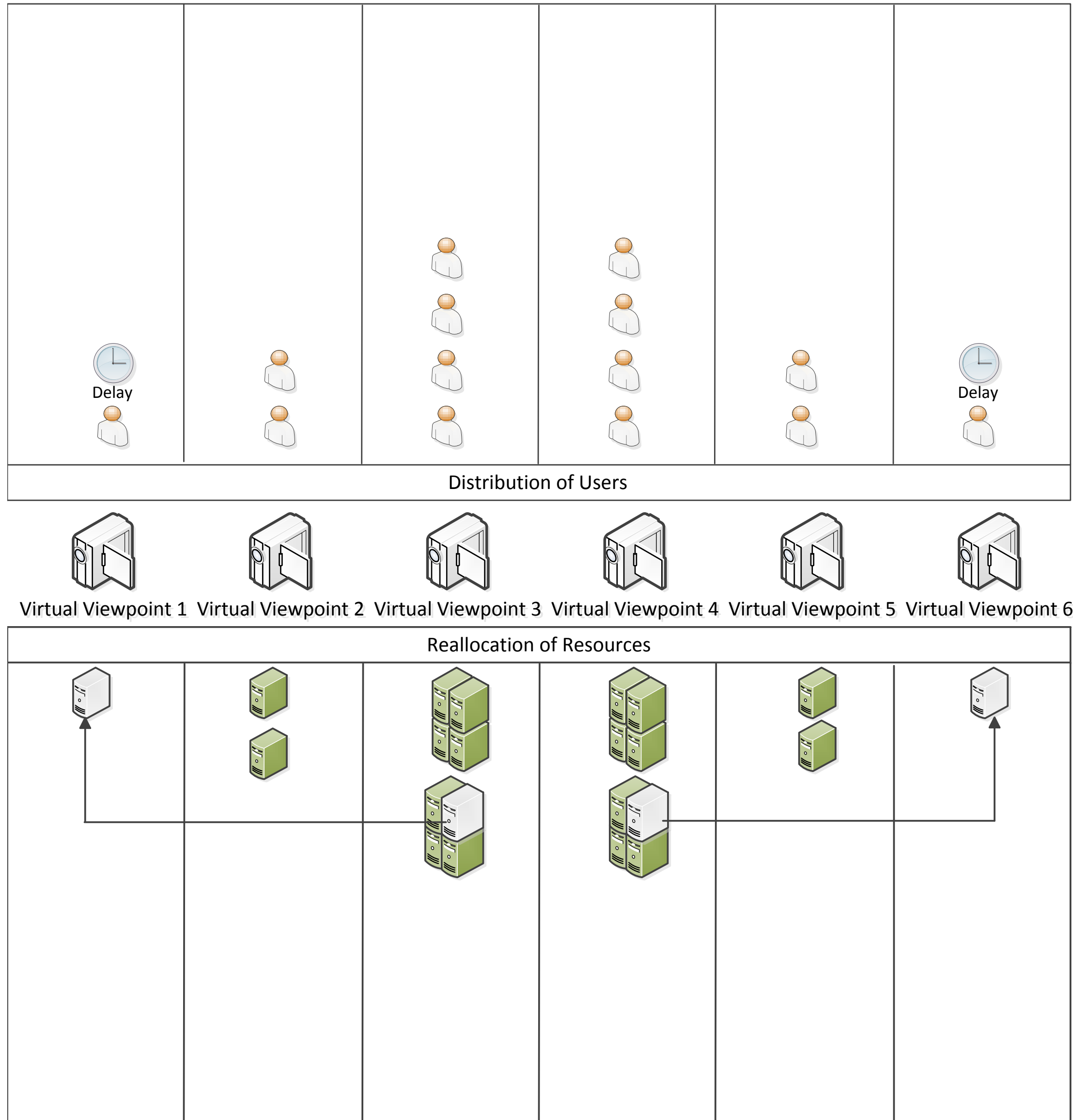


Figure 3.6 Reallocation of Resources among Viewpoints

According to our dynamic reallocation of resources, someone may ask if the VMs can be switched from sufficient points to insufficient points, why we do not provision them in the way as we reallocate at the beginning. The first reason is in some cases, there are

not enough VMs in total, we have to suffice the central viewpoints first, then support the other viewpoints. The second reason is the prediction cannot be always right. In some situations, some viewpoints may have over sufficient VMs while the others lack of VMs. In addition, there are some incidents like highlights unexpected during the streaming process. In particular, when highlight comes, most users will switch to specific viewpoints even they are not popular in the most of time. This kind of situations can be illustrated as Figure 3.7.

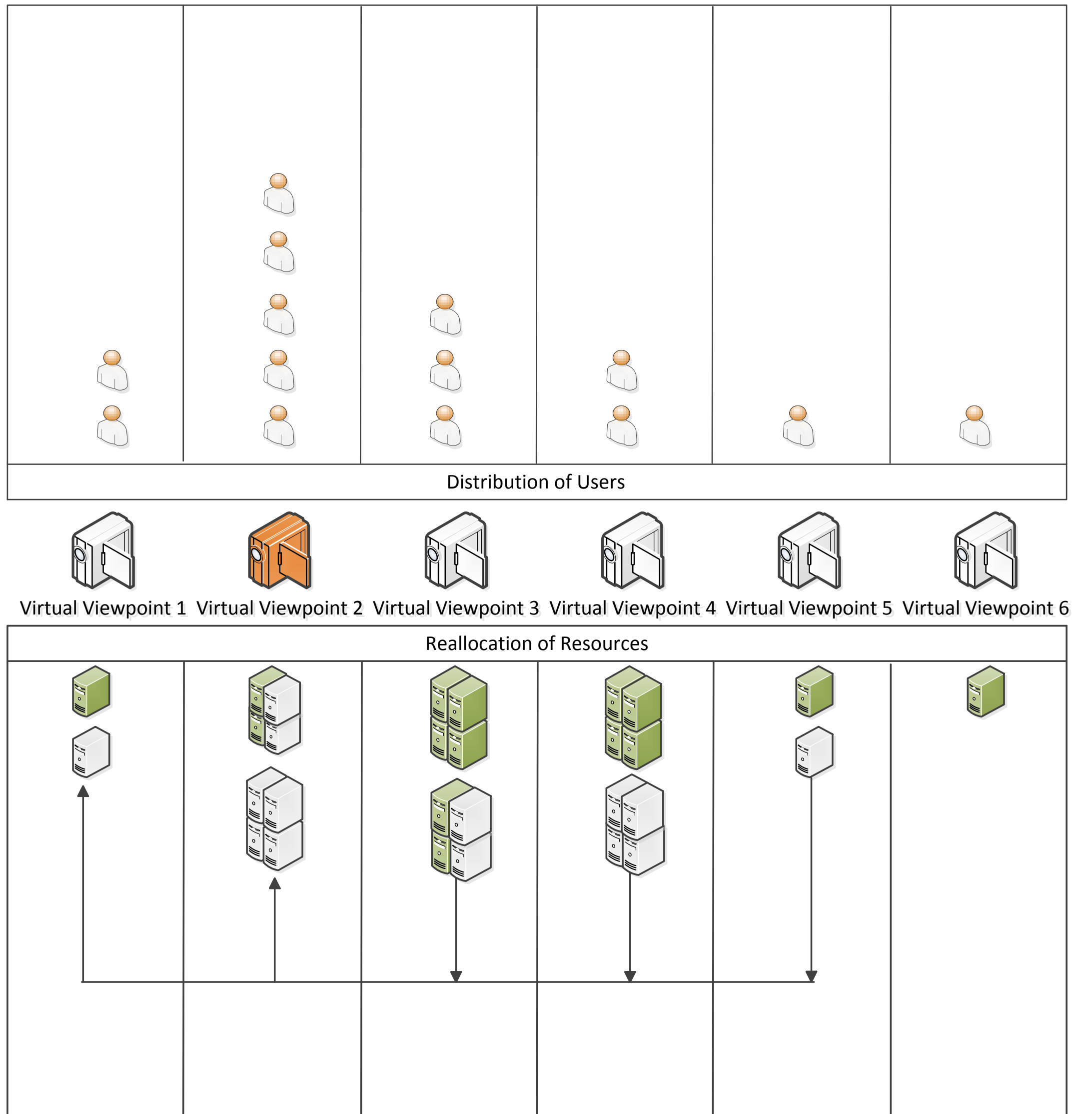


Figure 3.7 Dynamic Resource Allocation for the Highlight Viewpoint

In Figure 3.7, we suppose there is a highlight viewpoint which is virtual viewpoint 2. Since it is an unexpected situation, the provisioned VMs in viewpoint 2 are not sufficient

as users switch from other viewpoints. Accordingly, the VMs should also be moved to the highlight viewpoints to support from the other viewpoints. We can see the VMs are reallocated from virtual viewpoints 3, 4 and 5 to virtual viewpoints 1 and 2. Even the nearby viewpoint 1 can also have a sudden overload during this period of time. By this dynamic resource allocation scheme, we can maintain the relatively high quality of experience for most of users and deal with the possible incidents which result in overload on specific viewpoints.

Dynamic resource allocation is one of the key points in our real-time FVV rendering and streaming system. It can make the resources be used in the right place in a dynamic way. By this scheme, it is possible to reduce the economic cost without reducing much user experience at the same time. In addition, for each viewpoint provisioned with resources, we can apply our multithread processing strategy in order to further improve the performance and efficiency.

3.2.2. Multithread Processing Strategy

When a viewpoint is provisioned with cloud resources, it has the ability to produce FVV stream to the users. To speed up the production process and take full advantage of the resources, we introduce multithread processing strategy. The key idea in the strategy is split and merge. In this split and merge process, the most important point it to determine proper separation points.

In our multithread FVV synthesis process, we first decode the input video which is in texture and depth respectively. According to the DIBR, the processing is performed horizontally in frame. Thus, we can split a frame into several horizontal slices. Since the multiple threads within one process use the shared memory, these slices can be processed on multiple threads concurrently and the number of slices in a frame should match the number of available threads. After the FVV synthesis is done, we merge the slices back into complete frames and then encode them to the output FVV. Since the view synthesis is much more time consuming than the video coding, we only parallelize view synthesis in our system. When all frames are synthesised, we encode them together as traditional video at the end. The whole multithread processing workflow is illustrated in

Figure 3.8. The implementation of this multithread processing strategy will be introduced with details in Chapter 5.

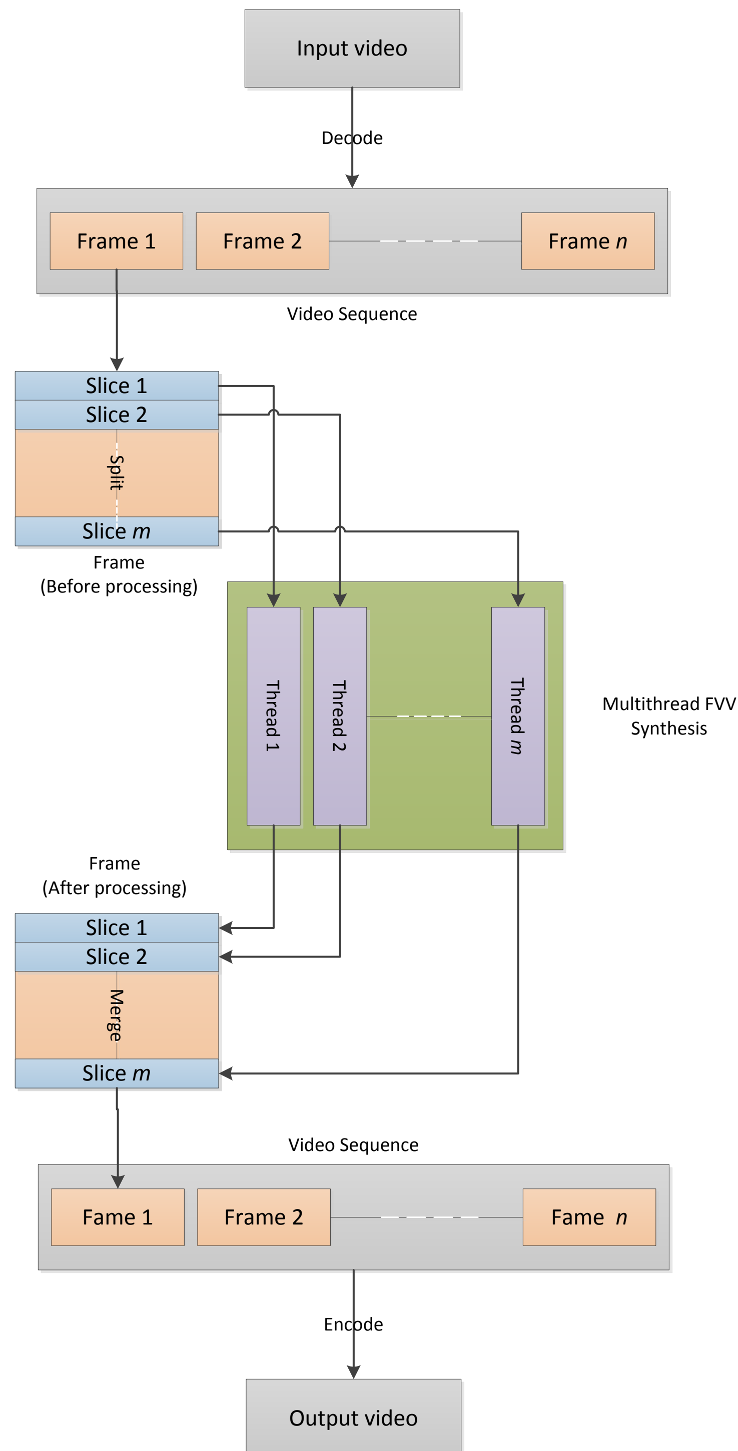


Figure 3.8 Multithread Processing Workflow

3.3. Summary

We introduce the architecture of our cloud-assisted FVV rendering and streaming system in this chapter. In the architecture, the capture side, cloud-side and client-side are integrated together through the Internet. As the core component, the cloud takes charge of receiving captured video, generating FVV, listening to users' requests and streaming FVV to users. To implement these functionalities, we design several modules for different purposes. For the control modules in the system, we design the dynamic resource allocation scheme and multithread processing strategy. These are the two key points in the whole system, since they determine how the resources are allocated and how the resources are used to generate FVV efficiently. However, in this chapter, we only describe the general logic. In order to achieve the optimal trade-off between economic cost and user experience, we still need precise modeling, calculation and algorithm. All these remaining concerns will be deal with in Chapter 4.

Chapter 4.

Mathmatical Modeling and Algorithm Design

In this chapter, the dynamic resouces allocation scheme and the multithread processing strategy are formulated in a mathmatical model for the precise calculation and result analysis. Based on the formulation and modelling, we introduce the objective function to quantify the trade-off between economic cost and user experience. In order to achieve the optimal solution, we propose a reliable algorithm which can be run in our system to make decisions. All the contents in this chapter focus on rendering and streaming live FVV in real-time by leveraging cloud resources. Before introducing the model and algorithm, we first describe the scenario and basic assumptions to narrow down the problem.

4.1. Basic Assumptions and Strategy

Nowadays, live video is becoming more and more popular. People can watch the video stream shortly after it is captured. To achieve the low latency live streaming, the video processing needs to be done in real-time. For the ordinary video, even the computer with common hardware can finish encoding and streaming in a very short time. However, since generating FVV requires processing of both texture and depth views, it is much more complicated and time consuming. Unlike non-live FVV which can be rendered in advance, the live FVV must be generated in real-time. Therefore, although it is challenging to realize, live FVV streaming is still a very practical scenario to use our real-time FVV rendering and streaming system. To support the real-time FVV synthesis, we integrate the rendering and streaming system with cloud computing. Here we give the basic assumptions based on this scenario.

First we narrow down the scope of this thesis to the cloud-side by making certain assumptions on capture side and client-side. Specifically, we assume the captured live videos are already sent to or cached in the cloud. So, the system can read the original views immediately. The number of available virtual viewpoints is m and all the viewpoints

can be denoted as $\{V_1,\ V_2, \cdots,\ V_m\}$. For the client, the users' requests are collected by the cloud every period of time and the number of users' requests for the viewpoint i during time slot t_u is M_{V_i,t_u} , where $V_i \in \{V_1,\ V_2, \cdots,\ V_m\}$. The users' requests are held in a queue until collected by cloud at the beginning of every time slot t_u .

For the cloud-side, the unit of resources is VM. We suppose that each VM has the ability to render FVV in real-time. But even if it is real-time, there is still a generating speed which depends on the quality of video, that is $Ge(Br)$, where Br is the bit rate of video indicating the quality. In addition, each VM has the upper bound of its serving bandwidth Bd, which will somehow determine the downloading speed of video streams and the maximum number of users it can serve at the same time. Due to the bottleneck of the input/output (IO) mechanism of VM, each VM can only run one task at a time. Therefore, we need to allocate VMs among all the viewpoints and each one of them can generate FVV for a specific viewpoint before it is reallocated. In addition, because of the bandwidth Bd and acceptable downloading speed $[Dl_L, Dl_H]$ of users, we can derive the number of the users which a VM can serve, that is

$$[Se_L, Se_H] = \left[\frac{Bd}{Dl_H}, \frac{Bd}{Dl_L}\right] \tag{4.1}$$

where Se_L, Se_H represents the lower bound and the upper bound of the number of users that a VM can serve. Within this range, a VM can provide all the users with acceptable downloading speed which determines the transmission delay.

In the cloud, the resources are reallocated as scheduled every t_c unit of time. For the reallocation, the system analyzes the trend and distribution of the users' requests within last several t_u time slots and then determines the updated reallocation scheme, including the number of VMs for each viewpoint in the next t_c units of time, that is N_{V_i,t_c} . Since reallocating all the resources takes certain time, we do not want this operation to happen too frequently. Thus, we assume that is $t_c \gg t_u$. However, in order to take full advantage of the cloud resources, we reallocate the VM immediately when it finishes tasks before the deadline within t_c. Specifically, when a VM is done with the current jobs but does not hit the end of t_c, it will fire an event to notify the listener in the control modules. Then the control modules will reassign it to other viewpoints where we need more VMs to

support. Once a VM is launched, the unit of its running time is the renting cycle T. It means that a VM will run at least T units of time because shutting it down before that is meaningless since the rental has already been charged at the beginning of the renting cycle. Usually the renting cycle of VM is relatively long, like 30 or 60 minutes. Therefore, we cannot immediately modify the existing renting plan until current renting cycle ends. When a VM finishes its current renting cycle, the system control can decide to shut it down or extend its renting period. Based on the workload situation of the previous renting cycle, we can also choose to launch more new VMs to serve users if the existing VMs are not sufficient.

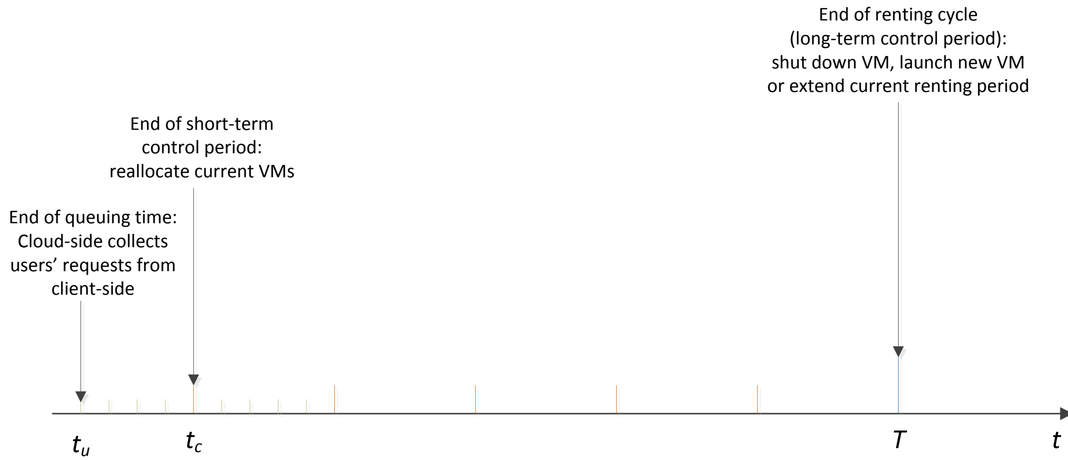


Figure 4.1 Timeline of one renting cycle

Basically, our dynamic resource allocation scheme can be divided into two parts, which are short-term and long-term control. For short-term control, it reallocates the current VMs at scheduled timing to take full advantage of their power. Although it does not shut VM down or launch new VMs, it has the ability to deal with the sudden overload on any specific viewpoints. For example, when there is highlight viewpoints and the distribution of users changes dramatically in a short period of time, the short-term control will force the reallocation without waiting to the scheduled timing. To determine if there is a significant overload on viewpoints which needs to force the reallocation immediately, we sort the users' requests at t_u when the cloud collects user requests, like

$$M_{V_1,t_u} < M_{V_5,t_u} < M_{V_2,t_u} < M_{V_4,t_u} < M_{V_3,t_u}$$

where V_3 represents the major viewpoint. If the order changes at $t_{u'}$ into

$$M_{V_1,t_{u'}} < M_{V_2,t_{u'}} < M_{V_3,t_{u'}} < M_{V_4,t_{u'}} < M_{V_5,t_{u'}}$$

where V_5 represents the highlight viewpoint, the short-term control should reallocate the VMs immediately otherwise the V_5 may experience significant overload. In summary, we allocate the VMs according to the order of the number of users' requests on each viewpoint. If the order changes at t_u, the system can reallocate the VMs immediately without waiting until the next t_c.

For long-term control, it can modify the resources provision plan by maintaining the total number of VMs in the system. In particular, it can turn off a VM, extend its renting period or add more VMs to the system. The timeline and corresponding events within one renting cycle are illustrated in Figure 4.1. While Figure 4.1 is explicitly showing the relationship between short-term and long-term control, the scale among t_u, t_c and T can varies much in the real world according to different charging schemes by the cloud service providers.

To make the right decision, the long-term control need consider three possible situations:

1. The current VMs are under sufficient to serve the users in the next renting cycle;
2. The current VMs are over sufficient to serve the users in the next renting cycle;
3. The current VMs are just sufficient to serve the users in the next renting cycle.

As a VM has the limitation on the number of users that it can serve at the same time, we can determine if the current VMs are enough to serve the users in the next renting cycle.

In the situation 1, the number of users exceeds the maximum which a VM can serve, that is

$$\sum_i M_{V_i,T'} > Se_H \cdot \sum_i N_{V_i,T} \quad (4.2)$$

where T is the current renting cycle and T' is the next renting cycle. Thus, we need to add more VMs in the next renting cycle; The situation 2 is the opposite of situation 1, that is

$$\sum_i M_{V_i,T'} < Se_L \cdot \sum_i N_{V_i,T} \tag{4.3}$$

In this situation, we can shut down some VMs to save the economic cost in the next renting cycle.

The situation 3 means that the serving capability of current VMs and the number of users are balanced, that is

$$\sum_i M_{V_i,T'} \in [Se_L, Se_H] \cdot \sum_i N_{V_i,T} \tag{4.4}$$

In this situation, adding more VMs will speed up the downloading but also increase the cost, while shutting down VMs will slow down the downloading but also reduce the cost. Besides, the system can also choose to extend the renting period of current VMs without increasing or decreasing the amount of VMs. Therefore, a decision should be made to optimize the trade-off, which depends on our economic cost and user experience models in the next section. And the decision making will be explained in detail in the section of algorithm design.

4.2. Cost and User Experience Model

In order to provide the optimal solution to the trade-off between economic cost and user experience, this section gives two models respectively and then defines the objective function to describe our ultimate goal.

4.2.1. Cost Model

The cloud resources are charged by the cloud service provider under certain scheme. And the charging scheme varies based on different kinds of VM with various hardware. To simplify this problem, we assume that we only rent one kind of VM which can generate FVV in real-time and this kind of VM is charged P dollars for a renting cycle

T. In addition, every viewpoint has its own economic value and it is usually different from each other, since the more popular and center viewpoint will charge more on the users. Thus, we represent it as Q_{V_i} for each viewpoint V_i . Assuming we only charge the users if they can watch the viewpoint without waiting time, so for each viewpoint we define a binary flag K_{V_i}, whose detailed definition is given in the next section as part of user experience model. Therefore, total cost C_{total} within one renting cycle is given by

$$C_{total} = \sum_i N_{V_i} \cdot P - \sum_i K_{V_i} \cdot M_{V_i} \cdot Q_{V_i} \quad (4.5)$$

4.2.2. User Experience Model

Unlike the economic cost model which is only affected by the price P of VM, the user experience model takes more aspects into consideration, such as FVV quality, FVV generating delay, transmission delay and waiting time. Basically, these aspects can be summarized into two categories, which are the FVV quality and the delay time until the users can watch the request FVV on their local devices.

For the FVV quality, we use Peak signal-to-noise ratio (PSNR) as measure. Basically, the PSNR is proportional to the bit rate Br of FVV but the detailed relationship is complicated. In the chapter 5, we can roughly describe it based on the experimental results. Here, we use a function to represent the FVV quality Q, that is

$$Q = PSNR(Br) \quad (4.6)$$

For the delay, we can formulate each part respectively. Firstly, although there is an assumption that the VM we rent can generate FVV in real-time, the generating speed can still be slightly different when using different bit rate setting. In order to maintain the real-time FVV generation and the user experience, the bit rate must be limited within an acceptable range, that is $Br \in [Br_L, Br_H]$. Supposing the users watch FVV for t time, the total generating time D_{Ge} should be

$$D_{Ge} = \frac{Pb \cdot t}{Ge(Br)} \quad (4.7)$$

where Pb also represents the playback rate of client-side.

Secondly, when the users are watching FVV rendered and streamed from the cloud, they need to download the streams, which will cause the transmission delay. The downloading speed depends on the bandwidth on both cloud-side and client-side. Generally, the users are assigned with certain network bandwidth from the network provider. Even though the cloud may provide larger bandwidth for the users, the downloading speed is still limited to an upper bound Dl_H. However, in some cases, the cloud cannot provide each user with the enough bandwidth to achieve the maximal downloading speed due to the large number of users' requests at the same time. Hence, although the users have sufficient bandwidth, the downloading speed is reduced. However, to maintain the real-time rendering and streaming experience, there is a lower bound of required downloading speed, that is Dl_L. Therefore, we can derive the formulation of the transmission delay on a specific viewpoint V_i,

$$D_{Tr} = \left(\frac{Bd \cdot N_{V_i}}{M_{V_i}}\right)^{-1} \cdot Pb \cdot t \quad (4.8)$$

Thirdly, as described in our dynamic resource allocation scheme, some viewpoints may not have provisioned VMs at the beginning. The users watching on these viewpoints have to wait for the other VMs finishing their tasks. There is an upper bound for the waiting time, because if the users wait for too long, the VMs will not have enough time to support them after being switched from other viewpoints. Thus, we can derive the condition on the waiting time D_{Wa},

$$\begin{cases} D_{Wa} = D_{Ge} \\ (T - D_{Wa}) \cdot Ge(Br) \geq Pb \cdot t \end{cases} \quad (4.9)$$

where T represents the total time before the deadline. Considering equation 4.7, we can conclude the condition as

$$D_{Wa} = D_{Ge} \leq \frac{T}{2} \quad (4.10)$$

which means the generating time cannot exceed half of the total time. Therefore, when we make provision for next renting cycle, if the $D_{Ge} > \frac{T}{2}$, we must add more VMs to the system, otherwise the FVV generation on certain viewpoints will miss the deadline. For each viewpoint, the derived waiting time is

$$D_{Wa} = K_{V_i} \cdot D_{Ge} \quad (4.11)$$

where $D_{Ge} \leq \frac{T}{2}$ and K_{V_i} indicates if the viewpoint has provisioned VMs. Assuming K_{V_i} indicates the viewpoints with sufficient provisioned VMs, it can be defined as

$$K_{V_i} = \begin{cases} 1, if \ \frac{M_{V_i}}{N_{V_i}} > Se_H \\ 0, otherwise \end{cases} \quad (4.12)$$

where

Here we make an assumption that a VM can be reallocated once within one timeslot. Hence, we have

$$\sum_i K_{V_i} \geq \frac{1}{2} \cdot m \quad (4.13)$$

where m is the total number of viewpoints.

Finally, in order to obtain the overall formulation of user experience, we consider the number of users $M_{V_i,t}$ on each viewpoint and use t to represent a timeslot. Thus, the overall average user experience (QoE) is

$$QoE_{average} = \alpha \cdot PSNR(Br) - t^{-1}(\beta \cdot D_{Ge} - \gamma \cdot D_{Tr} - \delta \cdot D_{Wa}) \quad (4.13)$$

that is,

$$QoE_{average} = \alpha \cdot PSNR(Br) - \beta \cdot \frac{Pb}{Ge(Br)} - \gamma \cdot \frac{\sum_i M_{V_i} \cdot \left(\frac{Bd \cdot N_{V_i}}{M_{V_i}}\right)^{-1} \cdot Pb}{\sum_i M_{V_i}} - \delta \cdot \frac{\left(1 - K_{V_i}\right) \cdot \sum_i M_{V_i} \cdot \frac{Pb}{Ge(Br)}}{\sum_i M_{V_i}}$$

$$(4.14)$$

where the conditions are

$$Br \in [Br_L, Br_H] \quad (4.15)$$

$$\frac{Pb}{Ge(Br)} \leq \frac{1}{2} \quad (4.16)$$

$$\sum_i K_{V_i} \geq \frac{1}{2} \cdot m \quad (4.13)$$

$$\frac{Bd \cdot N_{V_i}}{M_{V_i}} \in [Dl_L, Dl_H] \quad (4.17)$$

if $\frac{Bd \cdot N_{V_i}}{M_{V_i}} > Dl_H$, then shut VMs down; if $\frac{Bd \cdot N_{V_i}}{M_{V_i}} < Dl_L$, then launch more VMs. To further determine the number of VMs we need to shut down or launch, we need to integrate the QoE formulation to the economic cost formula to obtain the ultimate objective function.

4.2.3. Objective Function

In order to achieve the optimal resource allocation, the objective function should take both economic cost and user experience into consideration. To indicate the overall trade-off, the basic idea of objective function is to calculate the weighted average on these two factors, that is

$$S = \lambda \cdot C + (1 - \lambda) \cdot QoE \quad (4.18)$$

where S represents the score of the trade-off, C is the cost. However, the cost and QoE are in the different scales, we need to normalize them first to obtain the correct score.

For the cost, it is usually bounded by the budgets, that is $C \in [C_L, C_H]$. Thus, the normalized cost C^N should be

$$C^N = 1 - \frac{C}{C_H - C_L} = 1 - \frac{1/T \cdot \left(\sum_i N_{V_i} \cdot P - \sum_i K_{V_i} \cdot M_{V_i} \cdot Q_{V_i}\right)}{C_H - C_L} \quad (4.19)$$

where we use P/T to represent the cost of a VM per time unit. Since it is better to have lower cost, the C^N should be inverse proportional to the actual cost.

For the QoE, there are four internal factors which are PSNR, generation delay, transmission delay and the waiting time. To simplify the representation of the normalized score, we replace generation delay and transmission delay with generation speed and transmission speed respectively. As a matter of fact, the bit rate Br of FVV is limited to $[Br_L, Br_H]$, so the normalized PSNR is

$$PSNR^N(Br) = \frac{PSNR(Br) - PSNR(Br_L)}{PSNR(Br_H) - PSNR(Br_L)} \tag{4.20}$$

Similarly, we can derive the normalized generation speed

$$S_{Ge}^N = \frac{Ge(Br) - Ge(Br_L)}{Ge(Br_H) - Ge(Br_L)} \tag{4.21}$$

and the normalized transmission speed

$$S_{Tr}^N = \frac{\frac{\sum_i Bd \cdot N_{V_i}}{\sum_i M_{V_i}} - Dl_L}{Dl_H - Dl_L} \tag{4.22}$$

where Dl_H, Dl_L represent the range of possible transmission speed.

For the waiting time, the normalized score is also inverse proportional to the actual waiting time. In addition, based on the inference above, we can derive the range of the possible waiting time, that is

$$D_{Wa} \in \left[0, \frac{\sum_i^{m/2} M_{V_i} \cdot \frac{Pb}{Ge(Br)}}{\sum_i M_{V_i}}\right] \tag{4.23}$$

where $m/2$ means that at most half of the viewpoints has no provisioned VMs and Pb is a known parameter indicating the playback rate on client-side. In the most ideal case, the waiting time is zero as all the viewpoints are provisioned with VMs. Thus, the normalized waiting time is

$$D_{Wa}^{N} = 1 - \frac{\frac{\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}\cdot\frac{Pb}{Ge(Br)}}{\sum_i M_{V_i}}}{\frac{\sum_i^{\frac{m}{2}} M_{V_i}\cdot\frac{Pb}{Ge(Br)}}{\sum_i M_{V_i}}} = 1 - \frac{\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}}{\sum_i^{\frac{m}{2}} M_{V_i}} \tag{4.24}$$

To sum up, the normalized overall QoE is

$$QoE^{N} = \alpha \cdot PSNR^{N}(Br) + \beta \cdot S_{Ge}^{N} + \gamma \cdot S_{Tr}^{N} + \delta \cdot D_{Wa}^{N} \tag{4.25}$$

That is,

$$QoE^{N} = \alpha \cdot \frac{PSNR(Br)-PSNR(Br_L)}{PSNR(Br_H)-PSNR(Br_L)} + \beta \cdot \frac{Ge(Br)-Ge(Br_L)}{Ge(Br_H)-Ge(Br_L)} + \gamma \cdot \frac{\frac{\sum_i Bd\cdot N_{V_i}}{\sum_i M_{V_i}}-Dl_L}{Dl_H-Dl_L} + \delta \cdot \left(1 - \frac{\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}}{\sum_i^{\frac{m}{2}} M_{V_i}}\right) \tag{4.26}$$

Finally, we can obtain the overall trade-off formulation

$$S = \lambda \cdot \left(1 - \frac{1/T\cdot\left(\sum_i N_{V_i}\cdot P-\sum_i K_{V_i}\cdot M_{V_i}\cdot Q_{V_i}\right)}{C_H-C_L}\right) + (1-\lambda)\cdot\left(\alpha \cdot \frac{PSNR(Br)-PSNR(Br_L)}{PSNR(Br_H)-PSNR(Br_L)} + \beta \cdot \frac{Ge(Br)-Ge(Br_L)}{Ge(Br_H)-Ge(Br_L)} + \gamma \cdot \frac{\frac{\sum_i Bd\cdot N_{V_i}}{\sum_i M_{V_i}}-Dl_L}{Dl_H-Dl_L} + \delta \cdot \left(1 - \frac{\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}}{\sum_i^{\frac{m}{2}} M_{V_i}}\right)\right) \tag{4.27}$$

where P, T, C_H, C_L, Br, Br_H, Br_L, Bd, m are all parameters, Q_{V_i}, N_{V_i}, M_{V_i} and K_{V_i} are variables and $\alpha + \beta + \gamma + \delta = 1$.

Thus, our objective function is

$$\max_{Q_{V_i}, N_{V_i}, M_{V_i}, K_{V_i}} S\left(Q_{V_i}, \sum_i N_{V_i}, M_{V_i}, K_{V_i}\right) \tag{4.28}$$

$$s.t. \frac{M_{V_i}}{N_{V_i}} \in [Se_L, Se_H] \; and \sum_i K_{V_i} \geq \frac{m}{2}$$

where we can see that this optimization problem can be simplified into two basic sub-problems:

1. How many VMs should be rented according to the number of users;
2. How to allocate these VMs among all the viewpoints.

And the condition is the workload of a VM should be within the acceptable range. To solve these sub-problems, we will design an algorithm in the following sections.

4.3. Prediction Method and Resource allocation

According to our dynamic resource allocation scheme, the decision-making needs to depend on the estimation of users' requests in the next time slot. Hence, the accuracy of prediction method will determine the correctness of resource allocation. In this section, we present our prediction method first and then explain how the prediction results will affect our resource provision plan in the next time slot.

4.3.1. Short-term Prediction and Resources Reallocation

As design of our system, every t_u time the cloud-side collects the users' requests from client-side. Meanwhile, it can also obtain the distribution of users' requests on viewpoints, that is M_{V_i,t_u}. At t_c time, the scheduled resources reallocation happens and the system makes the resources provision plan for the next time slot. Supposing $t_c \gg t_u$, we can get a time series of users' requests on each viewpoint based on the previous time slots, that is

$$\left\{M_{V_i,t_{u_1}}, M_{V_i,t_{u_2}}, \cdots, M_{V_i,t_{u_n}}\right\} \tag{4.29}$$

where $t_{u_n} = t_c$ and $V_i \in \{V_1, V_2, \cdots, V_m\}$. In order to estimate the users' requests on each viewpoint in the next time slot $M_{V_i,t_{u_{n+1}}}$, we can leverage the usual prediction methods of time series, such as Autoregressive moving average (ARMA) and Autoregressive

integrated moving average (ARIMA). Since the prediction method is not our focus in this thesis, we just apply certain existing reliable method to our system. In Chapter 5, we will present the specific time series prediction method we use in the implementation of our system.

By leverage the time series prediction method, we can get the predicted number of users' requests on each viewpoint $M_{V_i,t_{u_{n+1}}}$. Based on this result, the short-term control will reallocate the VMs to each viewpoint. According to equation 4.27, the level of S only relies on the value of viewpoints and waiting time which is the last term, since the total number of VMs $\sum_i N_{V_i,t_{u_{n+1}}}$ will not be changed and users' requests on each viewpoint $M_{V_i,t_{u_{n+1}}}$ is known. Thus, the distribution of VMs K_{V_i} and the value of viewpoints Q_{V_i} are the factors which can affect the ultimate trade-off in the objective function.

According to the normalized waiting time, we can derive the simplified objective function

$$\max_{Q_{V_i},K_{V_i}} S' = \lambda \cdot \left(1 - \frac{\frac{1}{T}\cdot\left(\sum_i N_{V_i} \cdot P - \sum_i K_{V_i} \cdot M_{V_i} \cdot Q_{V_i}\right)}{C_H - C_L}\right) + (1-\lambda)\cdot\delta\cdot\left(1 - \frac{\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}}{\sum_i^{\frac{m}{2}} M_{V_i}}\right) \quad (4.30)$$

$$\text{s.t.} \frac{M_{V_i}}{N_{V_i}} \in [Se_L, Se_H] \ and \ \sum_i K_{V_i} \le \frac{m}{2}$$

where the possible maximum waiting time $\sum_i^{\frac{m}{2}} M_{V_i}$ is determined by M_{V_i}. The maximum waiting time happens when half of the viewpoints has no provisioned VMs and this half consists of the viewpoints with lower users' requests. Then, to maximize D_{Wa}^N, $\left(1-K_{V_i}\right)\cdot\sum_i M_{V_i}$ should be minimized. Hence, when the control system reallocates VMs, it should assign VMs to the viewpoints with most users' requests first and satisfy the condition $\frac{M_{V_i}}{N_{V_i}} \in [Se_L, Se_H]$. Since the magnitude of $\frac{M_{V_i}}{N_{V_i}}$ does not contribute to the ultimate objective, we can keep $\frac{M_{V_i}}{N_{V_i}} = Se_L$ during the reallocation process in order to avoid assigning over sufficient VMs on specific viewpoints. Once the viewpoint V_i with the most users' requests

has number of provisioned VMs $N_{V_i} = \frac{M_{V_i}}{Se_H}$, the system switches to the viewpoint with the second most users' requests, and so on. Through this greedy VMs reallocation strategy, we can finally achieve the optimal K_{V_i} and the maximum of the ultimate objective.

Based on the inference above, we can conclude that in order to achieve the optimal trade-off between economic cost and user experience, the priority on each viewpoint to obtain provisioned VMs should follow the order of number of users' requests. Last but not least, this short-term resources reallocation can be forced to run within a time slot when the order of number of users' requests changes. Since the short-term time slot is relatively short, the order of number of users' requests is usually stable during one time slot except when highlights appear suddenly in the video. By detecting the order of number of users' requests and activating the short-term resource reallocation accordingly, our dynamic resource allocation scheme is able to deal with the highlight viewpoints and relieve the influence from the fluctuation of the users' requests at the same time.

4.3.2. Long-term Prediction and Resource Provision

While the short-term predication happens at t_c to achieve the optimal resource reallocation scheme, the long-term prediction is triggered at the end of renting cycle T in order to decide if the number of VMs should be increased, decreased or maintained the same.

Similar to the short-term prediction, the long-term prediction is based on not only the distribution of users' requests, but also the total number of it, that is

$$\left\{\sum_i M_{V_i,t_{c_1}}, \sum_i M_{V_i,t_{c_2}}, \cdots, \sum_i M_{V_i,t_{c_n}}\right\} \tag{4.31}$$

where $t_{c_n} = T$ and $V_i \in \{V_1, V_2, \cdots, V_m\}$. Here we use the historical data on t_c within the previous renting cycle as the reference points to build the time series and assume the total number of users' requests cannot vary significantly within one renting cycle. By leveraging the time series prediction methods, we can obtain the predicted total number of users' requests within next renting cycle in addition to the distribution.

At the end of every renting cycle, the control system is able to change the total number of VMs. However, the lower bound of the total number is determined by the service capability of VMs and the number of users' requests. Thus, the starting point of total number of VMs is

$$\sum_i N_{V_i,T'} \geq \sum_i^{\frac{m}{2}} \frac{M_{V_i,T'}}{Se_H} \tag{4.32}$$

where T' represents the next renting cycle. Starting from this point, we can achieve the optimal solution by greedy approach. According to the objective function, the cost, transmission speed and waiting time will be affected by the number of VMs. Thus, we can derive the simplified objective function

$$\max_{Q_{V_i}, \sum_i N_{V_i}, M_{V_i}, K_{V_i}} \lambda \cdot \left(1 - \frac{\frac{1}{T} \cdot \left(\sum_i N_{V_i} \cdot P - \sum_i K_{V_i} \cdot M_{V_i} \cdot Q_{V_i}\right)}{C_H - C_L}\right) + (1-\lambda) \cdot \left(\gamma \cdot \frac{\frac{\sum_i Bd \cdot N_{V_i}}{\sum_i M_{V_i}} - Dl_L}{Dl_H - Dl_L} + \delta \cdot \left(1 - \frac{\left(1 - K_{V_i}\right) \cdot \sum_i M_{V_i}}{\sum_i^{\frac{m}{2}} M_{V_i}}\right)\right) \tag{4.33}$$

$$\text{s.t.} \frac{M_{V_i}}{N_{V_i}} \in [Se_L, Se_H] \ and \sum_i K_{V_i} \geq \frac{m}{2}$$

Specifically, in order to achieve the optimal result, we can keep trying to add one VM at a time to the system and allocate it according to the scheme with priority described in the previous section. After adding this VM, if the objective result is increased, we keep it and repeat the adding process; if the objective result is not increased, we add more than one VMs to the system to make sure reducing the number of viewpoints without provisioned VMs. Then if the objective result is still not increased, we discard these VMs and the number of VMs in the previous state is the optimal result. This greedy approach is illustrated in Figure 4.2. The solid line happens if the objective result increases, while the dashed line means falling back to the previous state if the objective result is not increased.

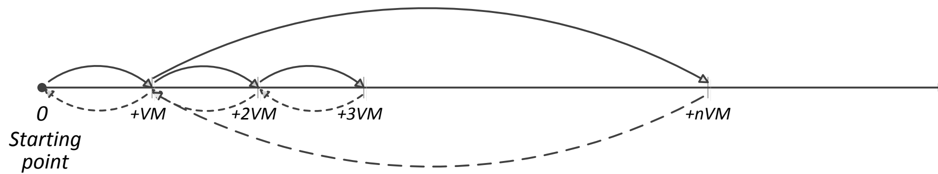


Figure 4.2 Resources provision starting from the lower bound

The above resources provision approach is suitable for the initial provision, since it can generate the optimal result from the lower bound. Considering the total number of users' requests between renting cycles usually changes little, the greedy approach can also start from the state of previous renting cycle. Accordingly, at the starting point, there are more options, which are removing VMs and maintaining the same number of VMs. Similarly, we can keep trying to remove the VMs but in the reversed priority order described in the previous section. If either adding or removing VMs cannot increase the objective result, the optimal solution is to keep the same number of VMs. Figure 4.3 depicts this approach.

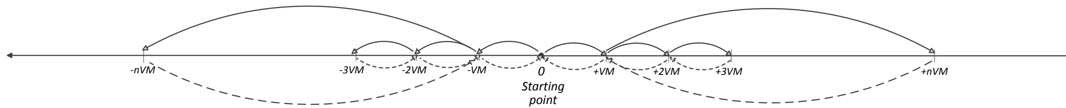


Figure 4.3 Resources provision starting from the state of previous renting cycle

4.4. Algorithm Design

Based on the previous sections, we can conclude the short-term resource reallocation algorithm and the long-term resource provision algorithm in Table 4.1, Table 4.2 and Table 4.2 respectively. The short-term resource reallocation algorithm has two versions. The first one only runs at the scheduled checkpoints ($t = t_c$) to reduce the overall computation consumption. The second one runs every time when the cloud-side collects the users' requests ($t = t_u$). It has better ability to deal with the fluctuation caused by

highlight viewpoints but introduces more computation consumption. These two versions can be selected according to different scenarios.

Table 4.1 Short-term resources reallocation algorithm (basic version)

	Input:
	The prediction of users' requests in the next time slot $\{M_{V_i}\}$ and the total number of VMs N.
	Output:
	The optimal VMs reallocation scheme $\{N_{V_i}\}$ in the next time slot.
1	**while** (true)
2	**if** $t = t_c$, **then**
3	sort $\{M_{V_i}\}$ in descending order on viewpoint V_i, that is $\{M_{V_i}\}^*$;
4	**foreach** $M_{V_i} \in \{M_{V_i}\}^*$
5	**while** $\sum_i N_{V_i} < N$ **AND** $M_{V_i} > Se_H \cdot N_{V_i}$
6	assign one VM to V_i, that is $N_{V_i} + 1$;
7	**end**
8	**end**
9	**if** $\sum_i N_{V_i} < N$, **then**
10	**foreach** $M_{V_i} \in \{M_{V_i}\}^*$
11	**while** $\sum_i N_{V_i} < N$ **AND** $M_{V_i} > Se_L \cdot N_{V_i}$
12	assign one VM to V_i, that is $N_{V_i} + 1$;
13	**end**
14	**end**
15	**end**
16	return the optimal reallocation scheme $\{N_{V_i}\}$;
17	**end**

Table 4.2 Short-term resources reallocation algorithm (with highlights solution)

	Input:
	The order of number of users' requests in the previous time slot $\{M_{V_i}'\}^*$, the prediction of users' requests in the next time slot $\{M_{V_i}\}$ and the total number of VMs N.
	Output:
	The optimal VMs reallocation scheme $\{N_{V_i}\}$ in the next time slot.
1	**while** $(t = t_u)$
2	sort $\{M_{V_i}\}$ in descending order on viewpoint V_i, that is $\{M_{V_i}\}^*$;
3	**if** $\{M_{V_i}\}^* \neq \{M_{V_i}'\}^*$, **then**
4	**foreach** $M_{V_i} \in \{M_{V_i}\}^*$
5	**while** $\sum_i N_{V_i} < N$ **AND** $M_{V_i} > Se_H \cdot N_{V_i}$
6	assign one VM to V_i, that is $N_{V_i} + 1$;
7	**end**
8	**end**
9	**if** $\sum_i N_{V_i} < N$, **then**
10	**foreach** $M_{V_i} \in \{M_{V_i}\}^*$
11	**while** $\sum_i N_{V_i} < N$ **AND** $M_{V_i} > Se_L \cdot N_{V_i}$
12	assign one VM to V_i, that is $N_{V_i} + 1$;
13	**end**
14	**end**
15	**end**
16	return the optimal reallocation scheme $\{N_{V_i}\}$;
17	**end**

Table 4.3 Long-term resources provision algorithm

	Input:
	The prediction of users' requests in the next time slot $\{M_{V_i}\}$ and the allocation of VMs in previous time slot $\{N_{V_i}\}'$.
	Output:
	The optimal VMs provision scheme $\{N_{V_i}\}$ in the next time slot.
1	**while** (true)
2	**if** $t = T$, **then**
3	choose one of the following:
	ADD: sort $\{M_{V_i}\}$ in descending order on viewpoint V_i, that is $\{M_{V_i}\}^*$;
	REDUCE: sort $\{M_{V_i}\}$ in ascending order on viewpoint V_i, that is $\{M_{V_i}\}^*$;
4	**foreach** $M_{V_i} \in \{M_{V_i}\}^*$
5	choose the one corresponding to the last choice:
	ADD: add one VM to V_i, that is $N_{V_i} + 1$, calculate the objective result S; REDUCE: reduce one VM on V_i, that is $N_{V_i} - 1$, calculate the objective result S;
6	**if** S is increased, **then**
7	repeat the step ADD/REDUCE;
8	**else**
9	choose the one corresponding to the last choice:
	ADD MORE: add VMs to V_i, such that $N_{V_i} \geq \frac{M_{V_i}}{Se_H}$, calculate S;
	REDUCE MORE: reduce VMs on V_i, such that $N_{V_i} = 0$, calculate S;
10	**if** S is still not increased **AND** only one of AND/REDUCE has been processed, **then**
11	switch ADD to REDUCE / REDUCE to ADD and repeat the process;
12	**else**
13	return the current provision scheme $\{N_{V_i}\}$ as the optimal solution;
14	**end**
15	**end**
16	**end**
17	**end**
18	**end**

4.5. Summary

Based on the dynamic resource allocation scheme, we develop mathmatical models in the chapter in order to calculate and analyse the trade-off between economic cost and user experience precisely. We use the normalized formulations to represent each factor which contributes to the overall trade-off and simplify the practical issue into a optimization problem. Through the greedy approach, we design algorithms to deal with short-term resources reallocation and long-term resouces provsion and develop the optimal solutions. With these algorithms, we can build our cloud-assisted FVV rendering and streaming system into practical software and test its performance in the next chapter.

Chapter 5.

Implementation and Experimental Results

In this chapter, we first introduce the implementation details of our cloud-assisted FVV rendering and streaming system, including the FVV synthesis software, video coding software, multithreading libraries, streamer development and cloud deployment. After that, we present typical experimental results generated from the prototype system and give out performance analysis based on that.

5.1. System Implementation

5.1.1. Free Viewpoint Video Synthesis

To generate FVV from the original views, we leverage a software module named view synthesis reference software (VSRS) [**15**]. The VSRS is an reference software for the 3D Video and FTV project of the 3D Video Coding Team of the ISO/IEC Moving Pictures Experts Group (MPEG). It was developed by Nagoya University, Thomson Inc., Zhejiang University, GIST, NTT, and TUT/Nokia in the course of development of the ISO/IEC JTC1/SC29 WG 11 (MPEG) 3D Video.

The VSRS provides us with C++ source code which can synthesize two original views into a novel virtual view based on the texture and depth information. In addition, a configuration file and a set of camera parameters are required to complete the FVV processing. Furthermore, the VSRS needs the Open Source Computer Vision (OpenCV) as a dependency in order to compile the source code and apply the functionalities. The VSRS supports both Windows and Linux. Here we use the Linux version in our implementation.

As an reference purposed software module, one of the major defects of the VSRS is its performance. Specifically, the VSRS performs video processing on both the texture and depth views. Since its processing unit is frame, the VSRS uses a loop to go through all the frames in the original views and write out synthesized frames one by one.

Apparently, this loop can be optimized on the system with multi-core processors using multithread programming technique. The sequence of frames can be divided into serval parts in order to be processed on multiple threads simultaneously. Another way is to split one frame into multiple horizontal slices, since the VSRS processes both texture and depth views horizontally. In our system implementation, we synthesize FVV in the latter way. To complete the implementation, we need to rewrite certain Application Programming Interfaces (API) of the VSRS. The details will be provided in the following sections.

5.1.2. Video Coding

Video coding is an important and necessary part in our system implementation, which can compress and encapsulate the raw video data into “streamable” video files. Nowadays, there are various video coding formats, such as H.264, HEVC. Here we choose H.264 as the codec used in our implementation. Video content encoded using a particular video coding format is normally inside a multimedia container format like AVI, MP4 and FLV. As such, the user normally doesn't have a H.264 file, but instead has a .mp4 video file, which is an MP4 container containing H.264-encoded video [**32**]. According to different video streaming protocols, there are correspondingly required video containers. For example, the RTMP requires FLV as the video container. Therefore, we need to apply specific video codec and container to the raw video data before streaming it over the internet. FFmpeg is such a software we leverage to complete this process.

As a free software project, FFmpeg provides libraries and programs for handling multimedia data. It includes an audio/video codec library, an audio/video container mux and demux library, and the FFmpeg command line program for transcoding multimedia files [**33**]. In order to integrate FFmpeg with our FVV synthesis software module, we use the FFmpeg libraries rather than the command line program in our implementation. The FFmpeg libraries have a large number of APIs to help with recording, converting and streaming audio and video. Through these APIs, we can also customize the settings of video codec including the bit rate, width/height, GoP size and Frames Per Second (FPS). Thus, in our experiments, we can easily compare the performance and results with different settings. In addition, on the cloud-side, we assume that the captured videos are already decoded and stored as YUV files. So, we can use the FVV synthesis module to

produce the videos on the novel virtual viewpoints and apply FFmpeg to encode them with suitable codec and container in order to stream them to the client-side.

5.1.3. Multithreading Libraries

Real-time is another focus in our system implementation. As the original VSRS has relatively low processing speed, it can hardly complete the tasks in real-time. Thanks to the multithread programming technique, it is possible to speed up the VSRS on the cloud with powerful computation resources. To take advantage of the cloud resources, we need to leverage multithreading libraries, such as POSIX Threads (Pthreads), Open Multi-Processing (OpenMP) and Message Passing Interface (MPI).

In shared memory multi-processor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as Pthreads [**34**]. As Pthreads is well supported in the UNIX systems like Linux and Mac OS, we use it to develop our multithread program on the VM running Linux. In addition, since the Pthreads is a light weight and low level multithreading library, it has high efficiency on the threads communication and data exchange as well as many sophisticated functionalities, including the mutex and condition variables. However, as a result, the Pthreads requires more programming effort in the implementation when compared to other tools.

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [**35**]. Compared to the Pthreads, OpenMP is more advanced and easily to use. It can quickly turn a single thread program into multithreading through a few statements in the code, especially for the loops in the program. Nevertheless, OpenMP only well supports to the loop-level parallelism and does not have complex

functionalities as the Pthreads. It has a limitation on the application scenario, which requires precise and complete threads control.

MPI is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran [**36**]. As a communication protocol for parallel programming, MPI's goals are high performance, scalability, and portability. Unlike the Pthreads and OpenMP, MPI can not only support shared memory system, but also the distributed system, which means MPI is not limited to the parallelism among multiple threads, but also the multiple distributed processes. Basically, MPI is a parallel computing approach based on both process and thread. It makes MPI very useful on the cloud cluster with distributed resource nodes, since the task can be divided into several processes to be performed on the multiple distributed nodes. Inside each process, we can use the Pthreads and OpenMP to make it multithreading to accelerate it further. Therefore, the combination of Pthreads, OpenMP and MPI should be the optimal solution to the real-time task on the cloud cluster with distributed nodes.

5.1.4. Video Streamer Development

The streamer is the last module in the cloud-side FVV rendering and streaming system. Basically, it handles the generated FVV and makes it into video stream in order to be delivered to the client-side. NGINX is a popular open-source and high-performance web server software which can be used to develop video streaming server. It was created by Igor Sysoev in 2002 and runs on various platforms, including Linux and Mac OS [**37**]. In addition, one of the major features of NGINX is that it supports the FLV and MP4 streaming. Thus, NGINX is an ideal software that we can apply to build our video streamer. To complete the implementation of the streamer, we also need a module to support RTMP/HLS live streaming, that is NGINX-RTMP-Module [**38**]. This module is open-sourced on Github [**39**] and widely used in live streaming server development. By leveraging the NGINX-RTMP-Module, our cloud-side can receive the captured video stream from cameras through RTMP and stream out the generated FVV to the client-side

through HLS. The users on the client-side can easily watch the live video stream using VLC [**40**] or the browsers with HLS support, like Safari. This module also provides elaborated streaming settings through the configuration file. So, we can customize the HLS fragments, playlist length to achieve the optimal watching experience for the users.

5.1.5. Cloud Deployment

After implementing all the modules in the FVV rendering and streaming system, the last step is to deploy them on a real cloud. In this thesis, we choose WestGrid as our cloud service provider. WestGrid is a government-funded infrastructure program started in 2003, mainly in western Canada, that provides institutional research faculty and students access to high performance computing and distributed data storage, using a combination of grid, networking, and collaboration tools [**41**]. WestGrid consists of several cloud clusters such as Breezy, Grex, Jasper which are located in the major universities in western Canada. Each cluster has different kinds and large scale of cloud resources which are set up as nodes. For example, Jasper [**42**] is a cloud cluster located in University of Alberta which has an aggregate 400 nodes with totally 4160 cores and 8320 GB memory. In the most of these nodes, there are 12 cores and 24 GB memory. It means that our application can be deployed on one node with at least 12 processing threads by using Pthreads and OpenMP. WestGrid also supports MPI for the large scale distributed computing. Thus, by leverage MPI combined with Pthreads and OpenMP, we can take full advantage of the power of cluster. To run our tasks on the cluster, WestGrid requires to submit a Batch Job Script to the cluster which can specify the amount of processors and memory needed. Then the submitted job will be placed into a queue waiting to be processed. Unlike Amazon Web Service (AWS), WestGrid does not allow us to rent and occupy individual resources like VMs or instances by personal applications, since it is research purposed and shared with many other users. But it is enough for us to test our system and collect experimental results. If the system needs to be deployed as a personal application and kept online throughout the day, AWS is a better choice with complete charge system on various instances with all kinds of hardware we need.

5.2. Experimental Results

5.2.1. Video Quality

As discussed in the previous chapter, the video quality is quantified as $PSNR(Br)$ which is determined by the bit rate of the synthesized FVV. In our experiment, we use a test video named “Balloon” along with the VSRS. Specifically, there are left and right views with both texture and depth information in resolution of 1024*768 which is shown in Figure 2.2. Through the VSRS we can synthesize the two original views and generate a novel virtual view in the middle. First, we compare the virtual view to the actual view captured at the same viewpoint by the real camera and both are not encoded. The result PSNR is 37.45, which means our generated FVV does not have the same quality as the actual view but it is still acceptable. Furthermore, we encode the generated FVV using different bit rate settings and compare them to the original FVV without encoding in order to figure out the relationship between PSNR and bit rate, that is $PSNR(Br)$. The PSNR calculation is done through a Matlab plug-in called YUV-PSNR which can compute the PSNR between two YUV files. The final results are shown in Table 5.1 and Figure 5.1. According to the results, we can see that the PSNR is increased dramatically as bit rate increases. In the following experiments, we can derive the $PSNR(Br)$ by referring to Table 5.1 and Figure 5.1.

Table 5.1 Relationship between bit rate and PSNR

Bit rate (bps)	PSNR (dB)
10M	45.04
5M	43.91
3M	43.00
2M	42.12
1.5M	41.37
1M	40.01
800K	39.31
700K	38.79

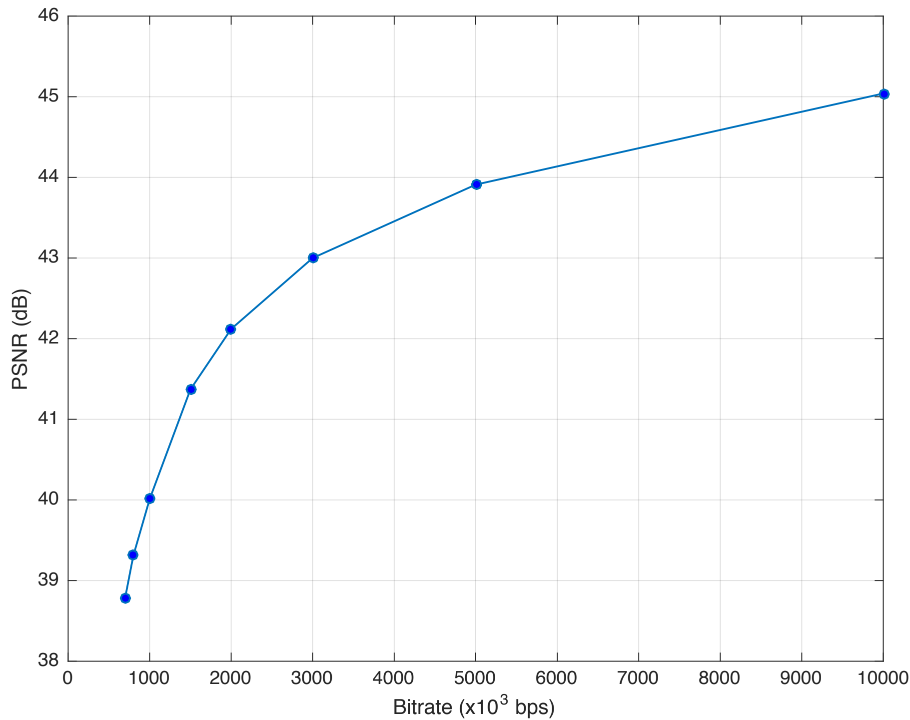


Figure 5.1 Bit rate and PSNR curve

5.2.2. Generation Speed

The FVV generation speed is another factor in our objective function which depends on the bit rate. To determine the relationship between them, we similarly performed experiments on the WestGrid cluster. Our testing environment is one node of Grex cluster [**43**] of Westgrid, which has 24 CPUs in total with shared memory. First, we investigate on the relationship between the number of processing threads and the number of vCPUs in order to determine the optimal ratio. The results are given in Table 5.2 and Figure 5.2.

Table 5.2 Relationship between threads and generation speed

Threads	Generation speed (fps)
1	2.77
2	5.23
6	14.21
12	20.63
24	27.57
48	30.58
72	31.32
96	31.98

Note. Test on a cluster node with 24 CPUs and the bit rate is set to be 1Mbps. The generation includes FVV synthesis and H.264 encoding.

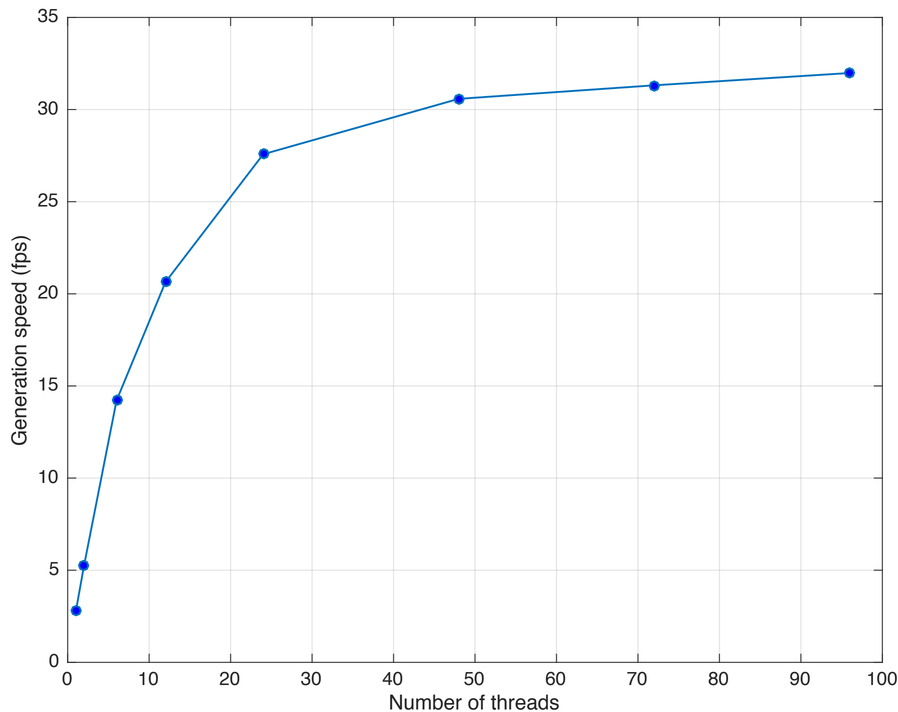


Figure 5.2 Number of threads and generation speed curve

From the results, we can conclude that the speed increases dramatically when the number of threads is less than the number of CPUs. However, after the number of threads reaches the double of CPUs, there is no significant improvement on the generation speed. It is because creating threads costs time and system resources. When the number of threads exceeds the double of CPUs, the time spent on initializing and switching threads

on cores becomes large enough to offset the acceleration of multithread processing. In the worst case, it might even delay the whole process. Therefore, the number of threads is usually set to be the double of the CPUs. Finally, when there are sufficient threads and CPUs, the generation speed can be over 30 fps which is enough for real-time streaming. It means that our optimization through multithread processing technique improves the speed of VSRS significantly from 2.77 fps to 30.58 fps and meet the requirement of real-time. To further investigate the relationship between the bit rate and generation speed, we figure out the speed with different bit rates in Table 5.3.

Table 5.3 Relationship between bit rate and generation speed

Bit rate (bps)	Generation speed (fps)
10M	29.41
5M	30.06
3M	30.33
2M	30.39
1.5M	30.51
1M	30.58
800K	30.61
700K	30.67

Note. Test on a cluster node with 24 CPUs and the number of threads is set to be 48. The generation includes FVV synthesis and H.264 encoding.

According to the result, we can see the higher bit rate may slightly slow down the generation process but does not make much difference on the overall generation speed. Even the lowest speed is over 29 fps which is still almost real-time (30 fps), which means that most of time cost is on FVV rendering, not the H.264 encoding. We can roughly estimate $Ge(Br)$ in our objective function based on the experimental results.

5.2.3. Users' Requests

The dynamic resource allocation algorithms proposed in the previous chapter all depend on the prediction of the distribution and number of users' requests in the future time slots. Since the prediction methods is not our focus in this thesis, we assume that the prediction results equal to the actual value in our experiment. Although the prediction accuracy is impossible to be 100% in real cases, the assumption is acceptable in the

experiment to test our algorithms on resource allocation according to users' requests. In addition, we suppose the users' requests follow Gaussain Distribution and total number of them at each t_u follows Poisson Distribution. The distribution and number of users' requests are assumed to be stable within one time slot t_u since t_u is usually very small in most cases. Here we set $t_u = 3$ seconds, $t_c = 60$ seconds, renting cycle $T = 15$ minutes and the number of viewpoints is 10 in the testing experiments. The distribution of users' requests among viewpoints is depicted in Figure 5.3. We set the central viewpoints as major viewpoints with higher distribution of users' requests. For the total number of users' requests at each t_u, we use Matlab to generate random numbers from the Poisson distribution with mean parameter lambda which is set to be 2000. The fluctuation of users' requests within one renting cycle T is shown in Figure 5.4.

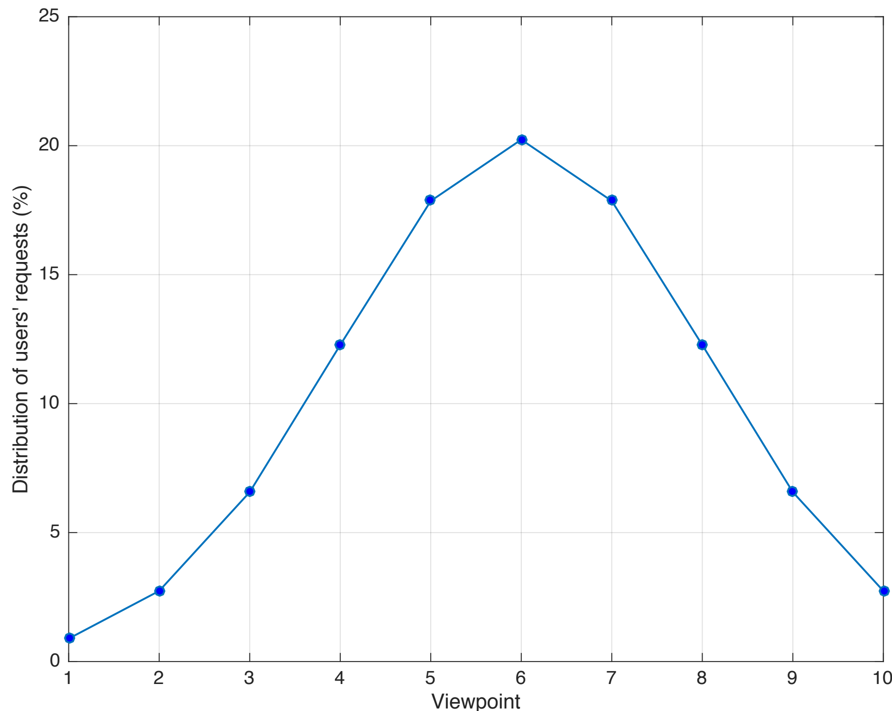


Figure 5.3 Distribution of users' requests among viewpoints

When integrating the total number of users' requests with its distribution, we can depict the distribution of users' requests on both time and space using a heat map, like Figure 5.5. As we have the simulation of users' requests, we can test the performance of our dynamic resource allocation algorithms and compare them to the normal static allocation scheme.

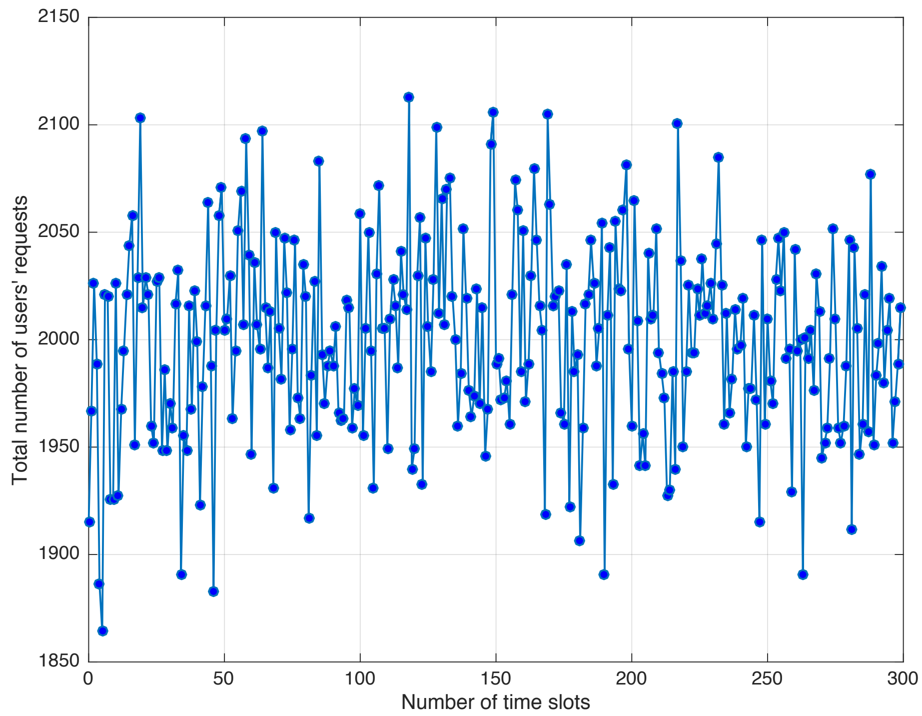


Figure 5.4 Users' requests within one renting cycle

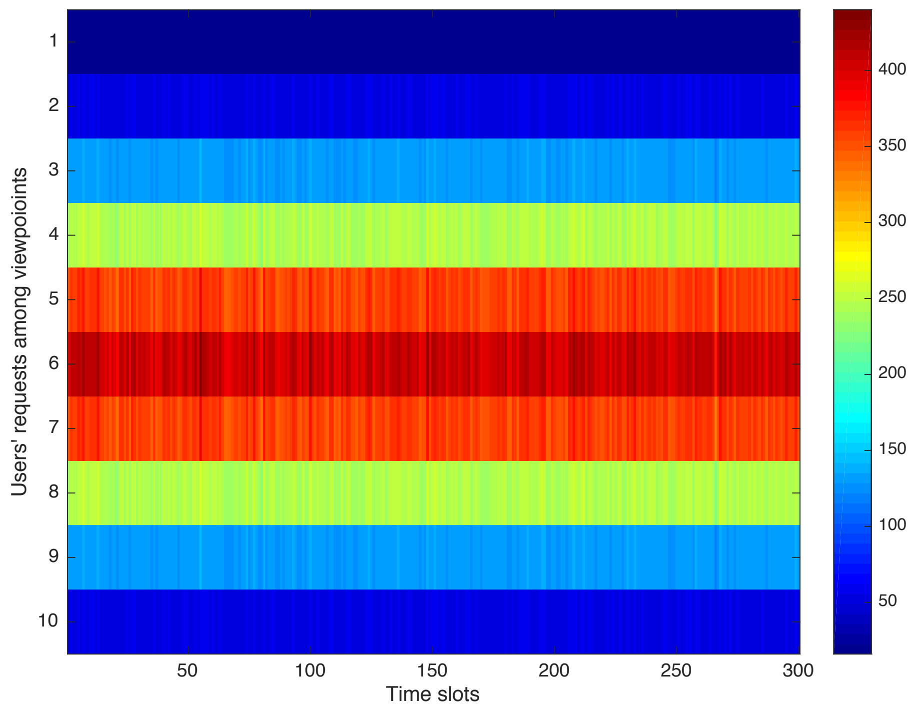


Figure 5.5 Distribution of users' requests within one renting cycle

5.2.4. Short-term Resources Reallocation

In order to test our short-term resources reallocation algorithm, we need the prediction of users' requests and the total number of VMs. For the prediction of users' requests, we use the simulation method which is described in the previous section and add a highlight period to compare the dynamic resources reallocation algorithm with the static one. Hence, the distribution of users' requests within one renting cycle can looks like Figure 5.6. Normally, the users' requests should concentrate on the central viewpoints like viewpoint 5, 6, 7 in Figure 5.6. However, sometimes highlight incidence will change the focus of users. For example, in the latter time slots, the users' concentration moves to viewpoint 2, 3, 4 from 5, 6, 7. Therefore, the resources should be reallocated accordingly through our short-time dynamic resources reallocation algorithm.

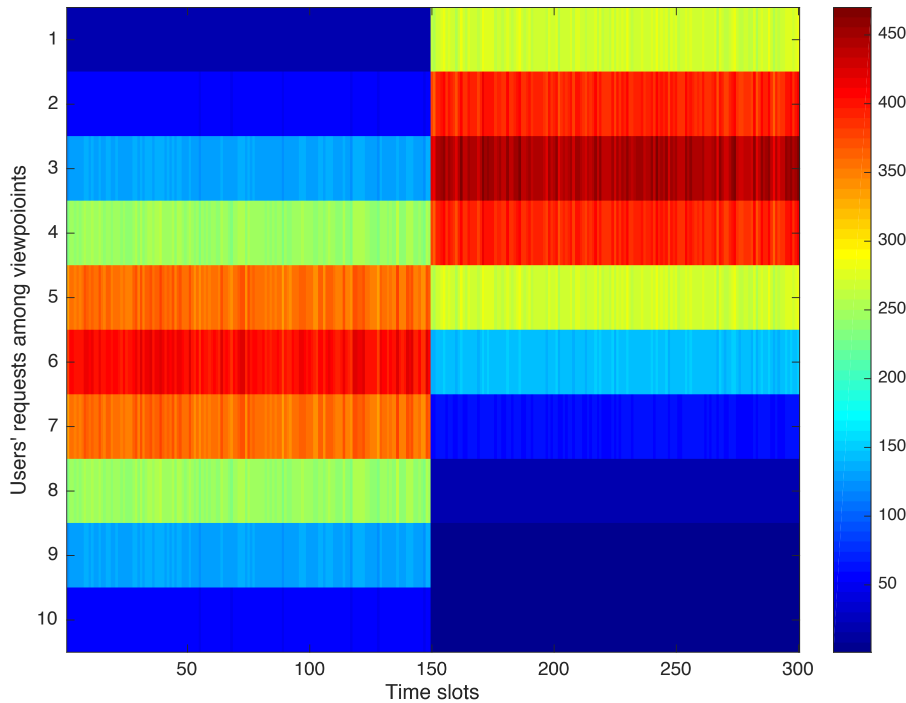


Figure 5.6 Distribution of users' requests with highlight in one renting cycle

Before testing the performance of algorithm, we first present the setting of testing parameters in table 5.4. Since we assume that the total number of VM cannot be changed within one renting cycle, we initialize the number of VMs among viewpoints in Table 5.5. According to the distribution of users' requests, if $V_i < Se_L$, we do not provision it with VMs; otherwise, we provision the viewpoints with sufficient VMs.

Table 5.4 Testing parameters in Equation 4.27

Parameter	Test value
Se_L, Se_H	100, 200
P/T	20
Q_{V_i}/T in the first half time slots	$\{1,2,4,6,8,10,8,6,4,2\}/100$ from V_1 to V_{10}
Q_{V_i}/T in the second half time slots	$\{6,8,10,8,6,4,2,1,1,1\}/100$ from V_1 to V_{10}
C_L, C_H	50, 250
λ	0.5
$\alpha, \beta, \gamma, \delta$	1,1,1,1
$PSNR(Br), PSNR(Br_L), PSNR(Br_H)$	40,38,42
$Ge(Br), Ge(Br_L), Ge(Br_H)$	30.5,30.4,30.6
Bd	100Mbps
Dl_L, Dl_H	0.5Mbps,1Mbps

Table 5.5 Initialized VMs among viewpoints

Viewpoint V_i	1	2	3	4	5	6	7	8	9	10
Number of VMs N_{V_i}	0	0	1	2	3	3	3	2	1	0
K_{V_i}	0	0	1	1	1	1	1	1	1	0

Afterwards, in order to determine the performance, we calculate the objective value S which indicates the overall trade-off between economic cost and user experience. At the middle point of time slot, the order of number of users' requests changes. So, our short-term resources reallocation algorithm is triggered. Table 5.6 shows the VMs distribution after reallocation. We also compare the performance of our dynamic resource allocation algorithm with static resource allocation in Figure 5.7. From Figure 5.7, we can see that when the highlight changes the focus of users among viewpoints at the middle of time slots, the objective value of static resource allocation drops over 20% while dynamic resource allocation only drops 5%, which means our algorithm has good performance dealing with the turbulence of distribution of users' requests.

Table 5.6 Reallocated VMs among viewpoints

Viewpoint V_i	1	2	3	4	5	6	7	8	9	10
Number of VMs N_{V_i}	2	3	3	3	2	1	1	0	0	0
K_{V_i}	1	1	1	1	1	1	1	0	0	0

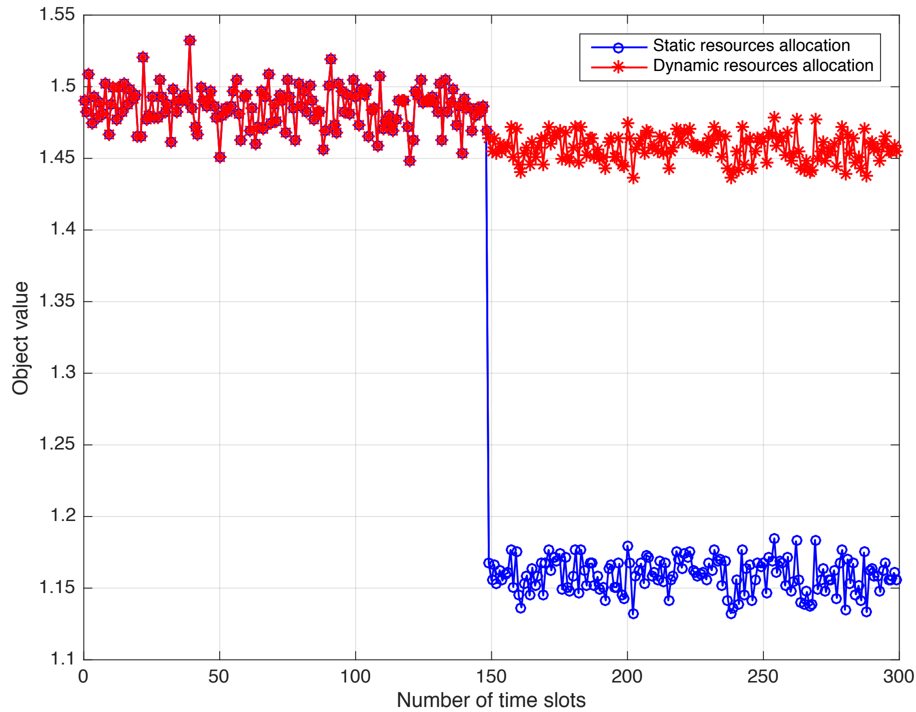


Figure 5.7 Comparison of dynamic and static resource allocation

5.2.5. Long-term Resources Provision

For our long-term resources provision algorithm, we can test it using the similar method in the previous section. Now we assume that the total number of users increases after the first renting cycle ends but the distribution remains the same. The trend of total number of users is illustrated in Figure 5.8 and Figure 5.9.

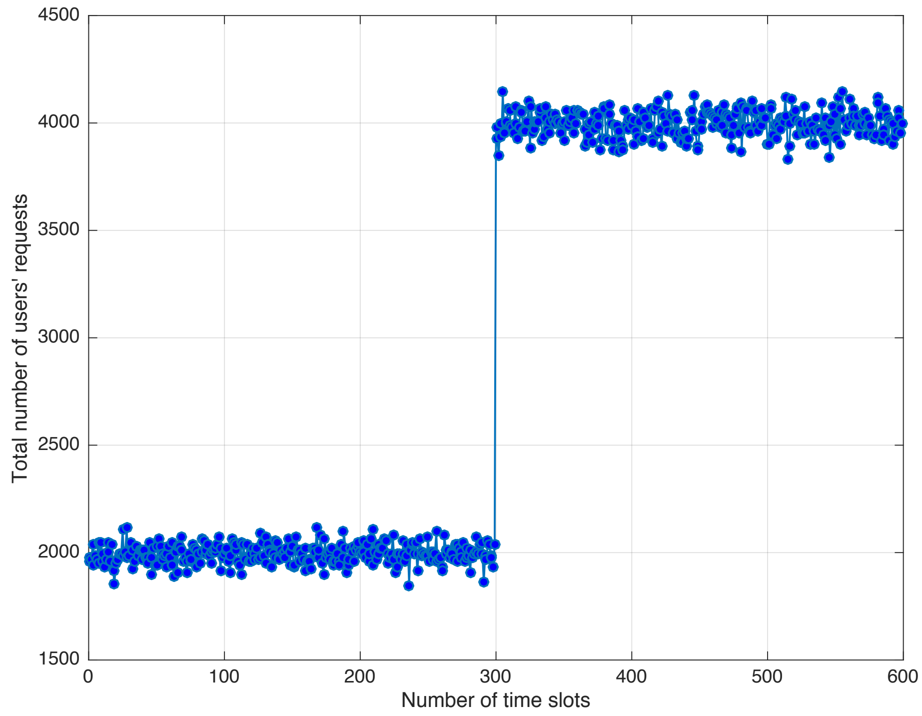


Figure 5.8 Total number of users' requests in two renting cycles

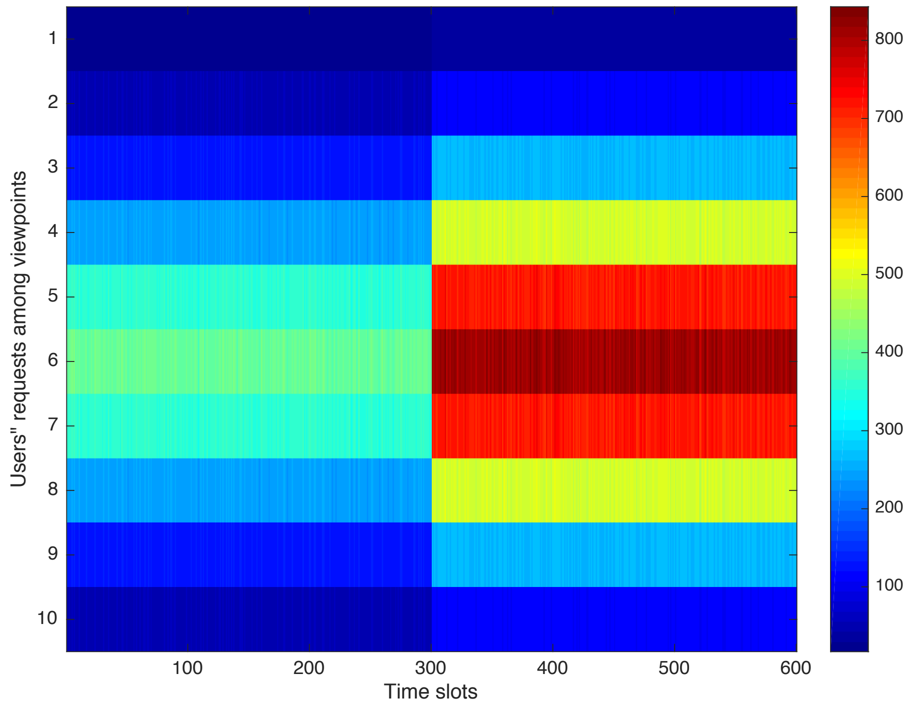


Figure 5.9 Distribution of users' requests with highlight in one renting cycles

At the end of a renting cycle, our long-term dynamic resources provision algorithm can help with determining how many VMs should be rented in the next following renting cycle. Supposing the VMs allocation in the previous renting cycle (time slot 0 to 299) is the same as shown in Table 5.5, our algorithm can figure out the optimal VMs provision in the next renting cycle (time slot 300 to 599) according to the increased total number of users' requests. The optimal results output from the algorithm are listed in Table 5.7. In addition, in Figure 5.10, we compare the optimal solution with another two cases which have one more VM and one less VM respectively.

Table 5.7 Provisioned VMs among viewpoints

Viewpoint V_i	1	2	3	4	5	6	7	8	9	10
Number of VMs N_{V_i}	0	0	0	3	4	5	4	3	0	0
K_{V_i}	0	0	0	1	1	1	1	1	0	0

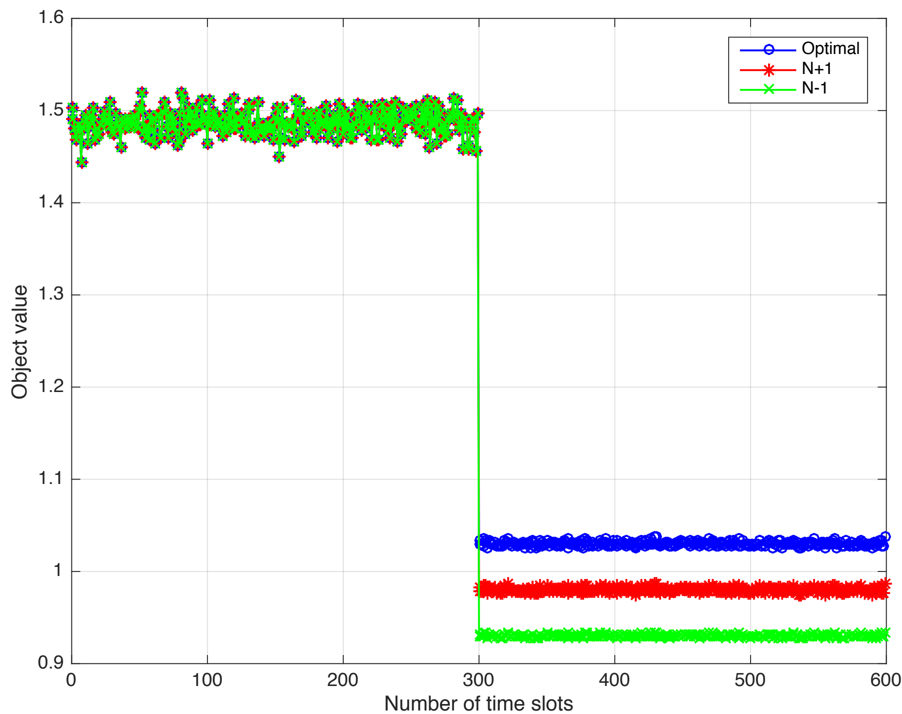


Figure 5.10 Comparison of the optimal solution and $N \pm 1$ cases

From Figure 5.10, we can conclude that our dynamic resources provision algorithm generates the optimal solution. If we add one more VM, it will increase the economic cost and result in lower objective value. Similarly, if we remove one more VM, it will cause the

streaming on one viewpoint to be shut down and result in lower objective value as well. In fact, we make basic assumption in the previous chapter that is to guarantee at least half of the viewpoints provisioned with VMs in order to avoid streaming on any viewpoint being closed due to lack of resources.

Chapter 6.

Conclusion and Future Work

In this thesis, we first introduce fundamental background of FVV rendering and describe the major techniques that we can leverage. Then we give an overview of cloud computing and explain how it can be applied to the real-time FVV rendering and streaming. Based on that, we design the whole architecture of our cloud-assisted FVV rendering and streaming system. For the three major parts in our system, which are video capture, cloud processing and client interaction, we describe them with details. After that, we demonstrate our approach to make FVV rendering real-time in the cloud. Both the resource allocation and task division are dealt with in Chapter 3.

Secondly, in Chapter 4 and 5, we formulate the dynamic resouces allocation scheme into mathmatical models. Base on the formulation and modelling, we propose the objective function to quantify the trade-off between economic cost and user experience. In order to ahcieve the optimal solution, we design algorithms to deal with both shor-time resouces reallocation and long-term resouces provision. After that, in Chapter 5, we use C++ multithread programming and Qt based on the VSRS, OpenCV, FFmpeg, Pthread/OpenMP/MPI and NGINX to complete implementation of our system. Then we deploy it on the WestGrid cluster to test with practical scenarios. Finally, our system is verified to be able to produce FVV stream in real-time and the trade-off between cost and use experience is optimal through our dynamic resource allocation algorithms.

This thesis presents an approach to make FVV rendering in real-time. However, it is based an assumption that is the network is ideal for all the users. In fact, according to various users' devices, the network condition can be very different. Thus, in the future work, we can take network into consideration and use variable rate codec for FVV encoding. Another future work focus can be on the cloud resources, since there are many different charging schemes on different cloud instances. In this thesis, we assume to use the identical type of instance. In the future, we can consider applying multiple types of resources in the system to achieve even better allocation scheme.

References

[1] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Free_viewpoint_television

[2] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Cloud_computing

[3] C. Luo, J. Wang and S. Li W. Zhu, "Multimedia Cloud Computing," *IEEE Signal Processing Magazine*, vol. 28, no. 3, pp. 59-69, May 2011.

[4] J. Starck, J. Kilner, and A. Hilton, "A Free-viewpoint video renderer," *Journal Of Graphics, GPU, And Game Tools*, vol. 14, no. 3, 2009.

[5] M. Volino, J. Guillemaut, S. Fenney, and A. Hilton J. Imber, "Free-viewpoint video rendering for mobile devices," in *the 6th International Conference on Computer Vision / Computer Graphics Collaboration Techniques and Applications (MIRAGE '13)*, vol. Article 11, New York, NY, USA, 2013.

[6] M. Levoy and P. Hanrahan, "Light field rendering," in *ACM Conference on Computer Graphics (SIG- GRAPH'96)*, New Orleans, USA, 1996, pp. 31-42.

[7] P.W. Rander, and P.J. Narayanan T. Kanade, "Virtualized Reality: Constructing Virtual Worlds from Real Scenes," *IEEE Multimedia 4(1)*, pp. 34-47, 1997.

[8] S. E. Chen and L. Williams, "View interpolation for image synthesis," in *the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*, 1993, pp. 279-288.

[9] C. Buehler, R. Raskar, S. Gortler, and L. McMillan W. Matusik, "Image-based visual hulls," in *ACM SIGGRAPH*, 2000, pp. 369–374.

[10] R. Grzeszczuk, R. Szeliski, and M. Cohen S. Gortler, "The lumigraph," in *ACM Conference on Computer Graphics (SIGGRAPH'96)*, New Orleans, USA, 1996, pp. 43-54.

[11] D. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. Salesin, and W. Stuetzle D. Wood, "Surface light fields for 3D photography," in *ACM Conference on Computer Graphics (SIGGRAPH-2000)*, New Orleans, USA, 2000, pp. 287–296.

[12] F. Ryan. Free Viewpoint Television. [Online]. http://ryansresearchproject.blogspot.ca/2010/03/free-viewpoint-television.html

[13] T. Popa, C. Zach, C. Gotsman, M. Gross C. Kuster, "FreeCam: A Hybrid Camera System for Interactive Free-Viewpoint Video," in *Vision, Modeling, and Visualization (VMV)*, Berlin, Germany, 2011, pp. 17-24.

[14] L. Do, P.H.N. de With S. Zinger, "Free-viewpoint depth image based rendering," *Visual Communication and Image Representation*, vol. 21, no. 5-6, pp. 533-541, July 2010.

[15] Krzysztof Wegner, Olgierd Stankiewicz, Masayuki Tanimoto, Marek Domanski, "Enhanced View SynthesisReference Software (VSRS) for Free-viewpoint Television," *ISO/IEC JTC1/SC29/WG11 MPEG2013/M31520*, October 2013.

[16] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *High Performance Computing and Communications, 10th IEEE International Conference*, 2008, pp. 5-13.

[17] and S. Ghemawat J. Dean, "MapReduce: simplified data processing on large clusters ," in *Communications of the ACM* , 2008, pp. 107-113.

[18] R. Pereira, M. Azambuja, K. Breitman and M. Endler, "An Architecture for Distributed High Performance Video Processing in the Cloud," in *IEEE 3rd International Conference on Cloud Computing*, Miami, FL, 2010, pp. 482-489.

[19] X. Nan, Y. He and L. Guan, "Optimal resource allocation for multimedia cloud based on queuing model," in *IEEE 13th International Workshop on Multimedia Signal Processing*, Hangzhou, 2011, pp. 1-6.

[20] D. Miao, W. Zhu, C. Luo, and C. W. Chen, "Resource allocation for cloud-based free viewpoint video rendering for mobile phones," in *the 19th ACM international conference on Multimedia (MM '11)*, New York, NY, USA, 2011, pp. 1237-1240.

[21] M. Zhao, X. Gong, J. Liang, J. Guo, W. Wang,X. Que, and S. Cheng, "A Cloud-assisted DASH-based Scalable Interactive Multiview Video Streaming Framework," in *31st Picture Coding Symposium*, Cairns, Australia, 2015, pp. 221-226.

[22] L. Toni, G. Cheung, P. Frossard, "In-Network View Synthesis for Interactive Multiview Video Systems," *IEEE Transactions on Multimedia*, vol. 18, no. 5, pp. 852-864, 2016.

[23] K. Kumar, J. Feng, Y. Nimmagadda and Y. H. Lu, "Resource Allocation for Real-Time Tasks Using Cloud Computing," in *20th International Conference on Computer Communications and Networks (ICCCN)*, Maui, HI, 2011, pp. 1-7.

[24] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading

[25] By I, Cburnett, CC BY-SA 3.0,. https://commons.wikimedia.org/w/index.php?curid=2233446.

[26] [Online]. http://www.perl.com/pub/2002/09/04/threads.html

[27] Howtogeek. [Online]. http://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/

[28] Tutorialspoint. [Online]. https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

[29] Amazon EC2 Pricing. [Online]. https://aws.amazon.com/ec2/pricing/?nc1=h_ls

[30] Wikipeida. [Online]. https://en.wikipedia.org/wiki/HTTP_Live_Streaming

[31] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol

[32] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Video_coding_format

[33] Wikipedia. [Online]. https://en.wikipedia.org/wiki/FFmpeg

[34] Blaise Barney, Lawrence Livermore National Laboratory. POSIX Threads Programming. [Online]. https://computing.llnl.gov/tutorials/pthreads/

[35] Wikipedia. [Online]. https://en.wikipedia.org/wiki/OpenMP

[36] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Message_Passing_Interface

[37] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Nginx

[38] Github. [Online]. https://github.com/arut/nginx-rtmp-module#nginx-rtmp-module

[39] Github. [Online]. https://github.com/

[40] VLC Media Player. [Online]. http://www.videolan.org/vlc/index.html

[41] Wikipedia. [Online]. https://en.wikipedia.org/wiki/WestGrid

[42] WestGrid, Jasper. [Online]. https://www.westgrid.ca/support/systems/Jasper

[43] WestGrid, Grex. [Online]. https://www.westgrid.ca/support/systems/Grex

[44] Steaming Learning Center. [Online]. http://www.streaminglearningcenter.com/articles/producing-h264-video-for-flash-an-overview.html?page=4