Tiling and Downsampling-based Immersive and Multiview Video Streaming Systems

by

Xiao Luo
B.A.Sc. (Electronics Engineering), Southeast University China, 2010

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

SIMON FRASER UNIVERSITY
Spring 2014

Approval

Name: **Xiao Luo**

Degree: **Master of Applied Science**

Title of Thesis: ***Tiling and Downsampling-based Immersive and Multiview Video Streaming Systems***

Examining Committee:

Chair: **Andrew Rawicz**
Professor

Jie Liang
Senior Supervisor
Associate Professor

Sami Muhaidat
Supervisor
Adjunct Professor

Jiangchuan Liu
Internal Examiner
Simon Fraser University, School of Computing Science
Associate Professor

Date Defended: February 21, 2014

Partial Copyright Licence

Abstract

Panorama photos/videos provide a brand new way to demonstrate 360 degree view of a scene in different platforms. While existing techniques can create and store the photos/videos as a whole frame, the first part of this thesis presents a development of efficient tile-based coding system for storing and transmitting the panoramic photos/videos, thus preserving the bandwidth and improving the quality of the photos/videos. It also improves the user experience especially for mobile users. Furthermore, the latest coding standard high efficiency video coding standard (HEVC) and its extension is utilized to replace H.264/AVC coding structure in original tile-based system. With a model-based rate distortion optimization algorithm for choosing the quantization parameters across different tiles and layers, we further improve the rate-distortion performance for this system. In the second part of the thesis, we focus on improving the efficiency of interactive multiview video streaming system, where users can switch view points during the playback, and the system can generate virtual view. A novel downsampling-based interactive multiview video stream system is proposed to offer a new degree of freedom to the system design. With the bitrate adaptive downsampling ratio selection and joint depth-texture bit allocation, the rate-distortion performance of the system can be improved at low rate regimes.

.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, professor Jie Liang, for his patience when I was in trouble with my research and his valuable guidance and advice. Besides, he gave me a lot of inspirations for my projects. I would also like to thank Dr. Andrew Rawicz and Dr. Jiangchuan Liu, Dr. Sami Muhaidat for their suggestions and time to review and improve my thesis.

Finally, an honorable mention goes to my families for their understandings and supports on me in completing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Social networks like Facebook, Twitter have attracted billions of users allowing people share their life online. Thanks to the ubiquitousness and convenience of mobile phones, the number of photos and videos uploaded to these social networks is exploding. For example, in the popular picture share application instagram [1], there are 16 billions photos, 1.2 billions likes daily, and 55 millions new photos each day. We can easily feel the trend that multimedia sharing will be dominant in social networks. This gives us the inspiration in two manners. One is that instead of traditional picture or video sharing, how to use new media formats to show our life. The other is due to the limit of bandwidth, how to save the bitrate consumption.

Panorama photos/videos and multiview videos are two emerging new media formats. A panoramic photo/video is basically a recording of a real world

scene from all angles, the viewer has control of the viewing direction, up down and sideways, where the view in every direction is recorded. Multiview videos provide a free viewpoint for the user to enjoy a 3D scene for movies or TV series. Applications like Google Street View, Immersive media [2] has shown that how powerful, useful and entertaining that panorama photos/videos can be. However, existing transmitting and storage schemes still have some space to improve.

The first part of this thesis focused on tile-based interactive panorama video system. In this part, we proposed an efficient way to transmit panorama video stream, where traditional whole panorama video frames are divided into multiple tiles. This system is inspired by ClassX which is an interactive online lecture video system developed by Stanford University. [3] As a result, with the combination of ClassX and OpenGL, a new tile-based mobile panorama video decoding system is created.

By pan/tile/zoom functionalities, this system allows users to easily get the region of interest (RoI). Therefore, the user experience is improved. Tile-based video streaming technique allows the system have a chance to satisfy the real-time interactive implementations. By further exploiting the rate distortion behaviour across different tiles, rate-distortion performance can be further improved. Up to 1.02 dB Peak Singal to Noise Ratio (PSNR) gain is achieved by this model-based approach.

Another problem for the existing tile-based system is that the correlation between different resolution layers has not been exploited. In this thesis, we

proposed a replacement of traditional H.264/AVC coding system by the scalable extension of the latest High Efficiency Video Coding standard, the SHVC. With this latest standard, we can further exploit the correlation between different layers and get more coding gain. In addition, by implementing spatial scalability bit allocation strategy from previous scalable video coding (SVC) standard, we again achieve up to 1.32 dB PNSR gain.

The second part of this thesis proposed a new strategy for coding interactive multi-view video stream (IMVS). This method is inspired by the fact that at low bitrate, it is more efficient to downsample the signal before encoding. With the same bitrate, we can achieve better R-D performance compared to traditional coding scheme. In [4], they proposed a mix-resolution strategy in multi-view video coding (MVC), in which the right-eye view is downsampled to improve overall visual quality. Brust [5] presented a mixed resolution approach for stereo video coding to get better subjective quality. However, the downsampling approach has not been applied to IMVS framework. In our approach, we assess not only the virtual views but also the base views. With this new framework, we also proposed a rate-adaptive approach to select the optimal down-sampling ratio according to the band-width budget. In addition to above contributions, we further study the bit allocation problem between depth and texture in IMVS. By proposed model-based joint optimization algorithm, we can get an improvement up to 0.65 dB gain in PSNR compared to traditional non-downsampling coding scheme.

1.2 Thesis Outline

Chapter 2 begins with the background of tile-based streaming system classX. The chapter details the rendering technique for current panorama photo/videos. Next, we explain how classX and OpenGL are combined.

Chapter 3 presents a model-based bit allocation algorithm for tile-based system. The idea is from scalable video coding bit allocation algorithm. We describe how this algorithm works in current framework and the fast algorithm is stated. Results are presented. After that, we consider transmission scenario, and use the same reasoning process with slight modification to get the solution.

Chapter 4 provides the background of SHVC and high efficiency video coding (HEVC), and details two distinct parts, the replacement for original H.264/AVC coding structure, and the optimization under SHVC framework, both of which are used in improving the video quality.

Chapter 5 proposes a novel down-sampling based IMVS approach. We then present the rate adaptive downsampling ratio selection strategy and the joint depth and texture bit allocation optimization is discussed.

Finally, the thesis is concluded in Chapter 6.

1.3 List of Contributions

The following demos and achievement have been produced during this project.

• Demo for OpenGL based ClassX decoder can be viewed in this link:

http://www.youtube.com/watch?v=JSq1TOGYBgo

• By model-based QP optimization algorithm for tiling based panorama video system, Up to 1.02 dB Peak Singal to Noise Ratio (PSNR) gain is achieved.

• By proposed downsampling based IMVS and model-based joint optimization algorithm for texture image and depth map, we can get an improvement up to 0.65 dB gain in PSNR compared to traditional non-downsampling coding scheme.

Chapter 2

Panorama Video Streaming System Based on Tiling Method

Thanks to the popularity of inexpensive high-definition (HD) video recoding technology, people can enjoy a detailed video when they are watching soccer games, Blue-ray DVD, and TV series. However, limited bandwidth or wireless network are often the hurdles for delivering HD video content. There are multiple ways to solve this problem. One way is to create HD video content at different quality levels. But this strategy will influence the user's experience and it does not consider the fact that people may just focus on some specific region of the whole frame.

2.1 ClassX Background

To solve above problem, Stanford University developed a novel interactive video streaming system called ClassX[3]. It is for publishing Stanford's lectures online. Since most existing streaming system only allows users to watch a predefined view, it is not convenient for users to select the region they are interested in. ClassX provides a tile-based video streaming system. With these feature, the video streaming system avoids sending the entire high definition (HD) frames, therefore reducing the total bits to be transmitted. Under this scheme, an encoder creates multiple resolution layers from original HD video sequence. For each layer, the video is divided into tiles. Each tile is encoded independently by H.264/AVC. The generated tiles are stored at the server. When the user select specific region of interest(RoI), the server will deliver the corresponding tiles. As a result, once the user has standard video decoders, they can easily get the RoI without transmitting the part they do not want. The base layer which has the lowest resolution will give users the whole video scene. This layer just has one tile which is called thumbnail and it is delivered to the user all the time. When a user zoom in the video, tiles in the higher resolution layers are retrieved and users will get better quality[3].

2.2 Panorama Video Rendering Background

Relying on the development of computer computation ability, three dimensional (3D) video/image processing technique is becoming more and more popular in

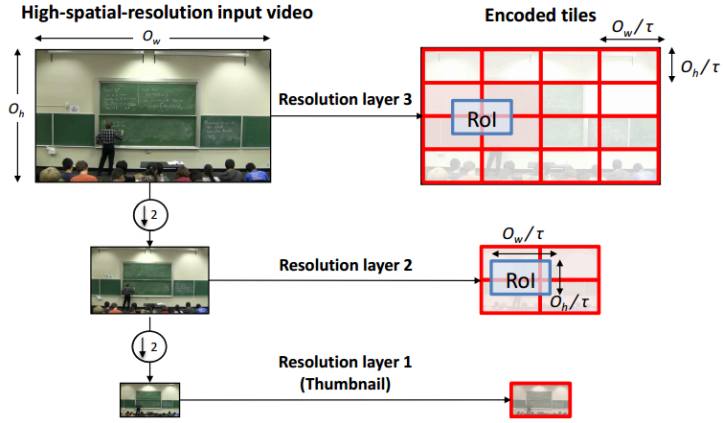


FIGURE 2.1: The coding scheme for classX. The original HD video is dyadically downsampled to three resolution layers. Each layer is further divided into tiles. Each tile is independently coded using and H.264/AVC video encoder

daily life, such as 3D video game and panorama TV. One of the most useful library and tool for realizing all kinds of 3D operations is OpenGL(Open Graphics Library). OpenGL is an application programming interface (API) for rendering 2D and 3D computer graphics on multiple platforms such iOS, android or Windows. With these APIs, we can interact with a Graphics processing unit(GPU), to achieve hardware-accelerated rendering[3].

With the help of OpenGL, we can map 2D video into 3D surface. Therefore, we can get a 3D virtual feeling of the image/videos. There are two kinds of existing techniques for storing 2D panorama image/videos. One is equirectangular projection, the other is cube mapping.

The equirectangular projection is a mapping that make a portion of the surface of a sphere to a flat image or its inverse operation. The mathematical expression

FIGURE 2.2: An equirectangular image from google street view

is as follows:[7]

$$\begin{aligned} x &= \lambda \cos \varphi_1, \\ y &= \varphi, \end{aligned} \tag{2.1}$$

where λ is the longitude of a sphere; φ is the latitude of a sphere; φ_1 are the standard parallels; x is the horizontal value for 2D texture coordinates; y is the vertical value for 2D texture coordinates.

Cube mapping is a mapping that uses a six-sided cube as the map shape[8]. The image is projected onto the six faces of a cube and stored as six square textures.

Cube mapping is usually preferred over equirectangular mapping because it eliminates many of the problems for equirectangular mapping such as image distortion, viewpoint dependency, and computational inefficiency. Also, cube mapping provides more flexibility to support real-time rendering.

With OpenGL, we can simply map flat image/videos to a sphere or a cubic and get our panorama image/videos.

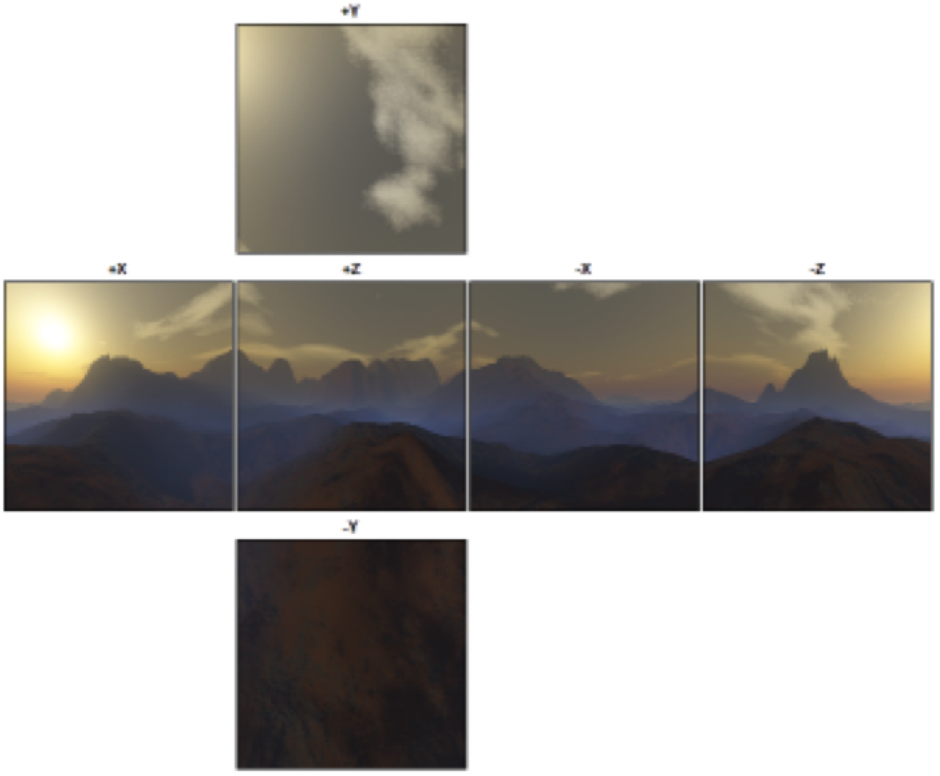


FIGURE 2.3: A cubic panorama image

2.3 Motivation and Implementation of OpenGL + ClassX

The motivation of this section is that rather than transmitting the whole panorama video stream, we use a tiling-based approach to save the bandwidth. To achieve this goal, the key point is to create a link between OpenGL and ClassX.

Because the rendering part is done by most open source application, like freepv [9] and Immersive Media, our focus is replacing their video coding module by ClassX. The platform we choose is iOS 5.0. With powerful Xcode 4.0 develop tools from Apple, we create a strong link between existing classX encoding structure and panorama rendering in OpenGL. To be more specific, we combine

ClassX coding system with typical rendering environment OpenGL, a cubic mapping demonstration is generated in our demo. The implementation detail is in Appendix A[10].

Fig 2.4 demonstrate our software which links OpenGL and ClassX. In the cubic rendering video (c) and (d), it can be seen that the video is now in the cubic texture rendering which means that we can apply OpenGL operation to this video from now on. And also, when users presses + or - button, the texture become clear gradually which means the stream right now is different from before. In another word, when the user zoom in the video, the tiles that is delivered is in the second or third resolution layer.

Note that in the demo, the video is directly streamed from the ClassX server, not from the local computer. In conclusion, in this chapter, we successfully created is an OpenGL-based ClassX decoder.

FIGURE 2.4: Illustration for the demo for OpenGL + ClassX

Chapter 3

Bit Allocation for Tile-based Interactive Panorama Video

3.1 Model-Based Bit allocation Scheme

Previous chapter introduces the tile-based panorama video streaming system. It allows a panorama sequence to be divided into some rectangular blocks which are called tiles. In the classX, fixed quantization parameters (QP) is assigned to each tile. In general, this is not optimal. This assignment is only suitable for the sequence with similar content throughout the whole frame. However, tiles of a frame can usually have different contents. Some of the tiles need more bits to achieve the same quality (PSNR), some of them need less. Inspired by this fact, we proposed to optimize the QP for each tile, based on the rate-distortion (R-D) characteristic for each tile. Using this model-based bit rate allocation algorithm, the overall R-D performance of the whole sequence is improved. In addition, in real streaming system, we can get the probability distribution for

the users' selection for each tile from the server. So it is desirable to develop an optimized bit allocation scheme by taking the tile content difference and the selection probability patterns into account. We put this probability issue into our problem formulation when doing the optimization.

Motivated by the same reason, Singara[12] proposed an entropy based weighted bit rate allocation algorithm. This algorithm use the fact that entropy of a complex tile is more than the entropy of smooth tile. It has a good performance but the complexity is too high. Zhicheng Li[13] proposed a visual attention-based bit allocation strategy to improve the subjective quality of the video. However, it is not applicable to tile-based panorama video system.

The motivation for this section is to provide an algorithm to improve the video quality and meet the constraint of certain bandwidth by selecting appropriate quantization parameters. The key issue in bit allocation and rate control is to estimate or model the R-D behavior of the video encoder. The R-D behaviour of an encoder is specified by its rate-quantization (R-Q) and distortion-quantization (D-Q) functions. In order to get the desired performance, accurate R-Q and D-Q models are the key point.

If we regard every tile as a layer in Scalable Video Coding (SVC), we can reference to some existing bit allocation schemes in SVC[11]. Unlike the layers in SVC which are dependently encoded, each tile here is encoded independently and we do not need to consider the dependency in this chapter.

3.1.1 Problem Formulation

We assume that the bit rate constraint for one group of picture (GOP) of a sequence is given. So within one GOP, the quantization parameter (QP) will be assigned to each tile. In another word, during the period of one GOP, for specific tile, the QP will be fixed. However, different tiles can get different QP. When a bit budget constraint is imposed, it is essential for an encoder to efficiently distribute the bit budget to tile for the optimal coding efficiency.

With the selection pattern collected from the users, we can get the optimal QP selection for each tile. In this section, we focus on the bits stored at the server, i.e, we use total bit rate here. In section 3.3.1, we will use the expected bit rate to simulate the actual transmitted bits.

Let N be the number of tiles. $R_k(Q_k)$ and $D_k(Q_k)$ are the distortion and rate model of the k-th tile with respect to a quantization vector$(Q_1, ..., Q_k)$. Given the bit budget R_{total}, the bit allocation problem can be formulated as

$$\begin{aligned} Q^* &= (Q_1^*, ..., Q_N^*) = \arg\min_{Q_k \in Q} \sum_{k=1}^{N} w_k \cdot D_k(Q_k) \\ s.t. &\sum_{k=1}^{N} R_k(Q_k) \leq R_{total} \end{aligned} \tag{3.1}$$

where $Q^* = (Q1^*, ..., Q_k^*)$ is the selected Q vector for all tiles. Q is the set of all quantization candidates. and w_k is the probability for the users' selection of each tile. As a result, the total distortion is defined as a weighted sum of each individual tile.

The Lagrangian multiplier method converts the constrained optimization problem in eq. 3.1 to an equivalent unconstrained optimization problem by creating the cost function as

$$\begin{aligned} Q^* &= \arg\min_{Q_k \in Q} J(Q, \lambda), \\ J(Q, \lambda) &= \sum_{k=1}^{N} w_k \cdot D_k(Q_k) + \lambda \sum_{k=1}^{N} (R_k(Q_k) - R_{total}) \end{aligned} \tag{3.2}$$

where λ is the Lagrangian multiplier.

To solve the quantization step size Q in eq. 3.2, one solution is to use a full search method over all possible combinations. However, since complexity is exponentially large when the number of tiles increases, we need alternatives without decreasing the performance too much. To solve this problem, we present a model-based solution in next two sections.

3.1.2 Distortion and Rate Modeling

Generally speaking, the R-D characteristics of a tile are represented by a function consisting of quantization step sizes. The impact of an individual quantization parameter on the R-D characteristics has to be known to solve the bit allocation problem.

For the R-Q model and D-Q model, we employ the models developed in [11].

$$\begin{aligned} D(Q_i) &= b \cdot Q_i^{\beta}, \\ R(Q_i) &= a \cdot Q_i^{-\alpha} \end{aligned} \tag{3.3}$$

Where Q_i is the quantization step, a, b, α, β are model parameters. For the derivation of model parameters, we first plot R(Q) and D(Q) for every pre-encoded tile, then use curve fitting tools in method find out the optimal parameters for each tile. As shown in figure 3.1, we can have nearly 90% accuracy to model the R-D behaviour of each tile.

3.1.3 Solution to the Lagrangian Formulation

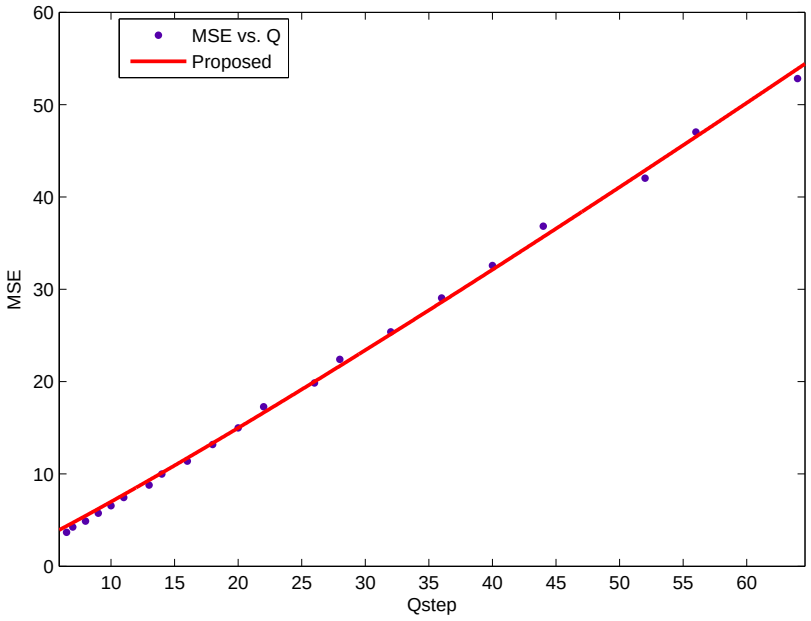
With the R-Q and D-Q models and the parameters derived above, we have everything we need to solve the bit allocation problem for tile-based video streaming system. Since the proposed models are defined by closed-form expressions, a numerical solution to the Lagrangian formulation becomes doable.

By inserting the R-Q and D-Q models expressions, the Lagrangian cost function can be written as

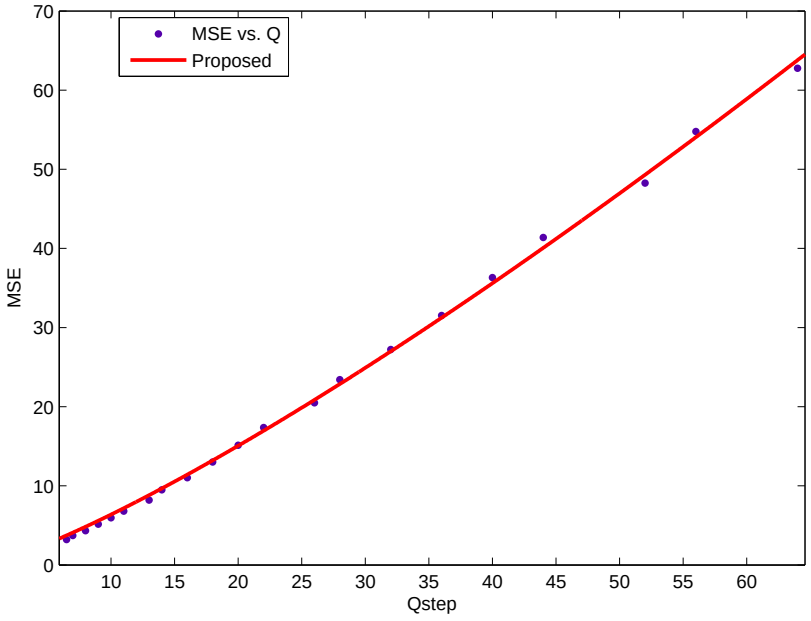
$$\begin{aligned} J(Q,\lambda) &= \sum_{k=1}^{N} w_k \cdot D_k(Q_k) + \lambda(\sum_{k=1}^{N} R_k(Q_k) - R_{total}) \\ &= \sum_{k=1}^{N} w_k \cdot b \cdot Q_k^{\beta_k} + \lambda(\sum_{k=1}^{N} a \cdot Q_k^{-\alpha_k} - R_{total}) \end{aligned} \tag{3.4}$$

To derive the optimal solution of the Lagrangian cost function, we take the partial derivatives with respect to Q_k and λ, which yields the following equations:

$$\begin{aligned} & w_k \cdot b\beta_k Q_k^{\beta_k - 1} + \lambda \cdot (-\alpha_k a Q_k^{-\alpha_k - 1}) = 0, k = 1..N, \\ & \sum_{k=1}^{N} a \cdot Q_k^{-\alpha_k} - R_{total} = 0 \end{aligned} \tag{3.5}$$



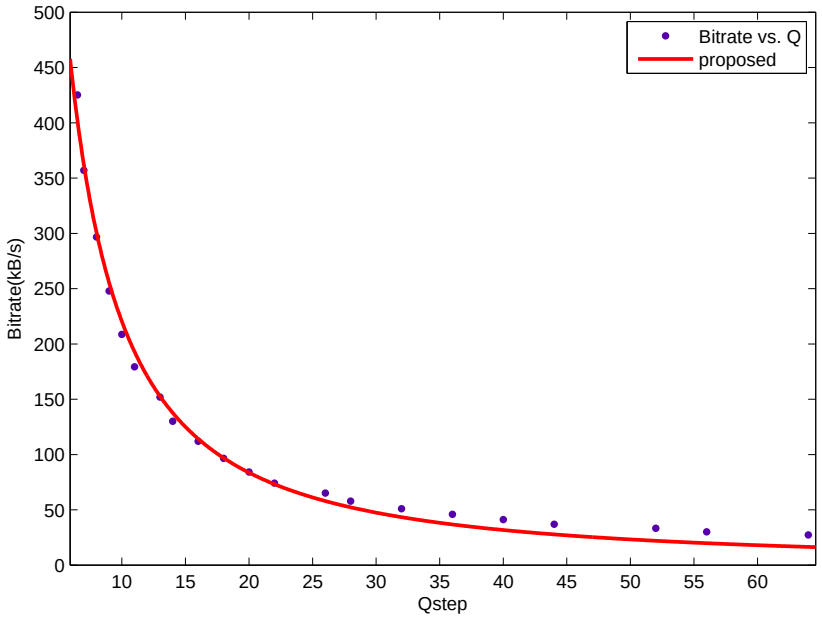
(a)



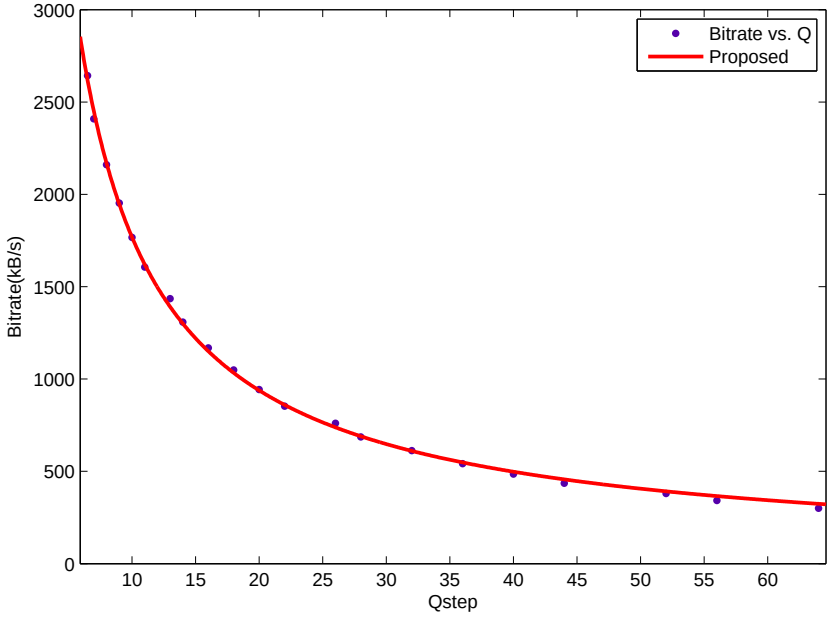
(b)

FIGURE 3.1: Curve fitting result for the proposed D-Q model

Note that there are $N + 1$ variables Q_1,Q_2 and λ to be solved because other parameters are derived in an earlier stage.



(a)



(b)

FIGURE 3.2: Curve fitting result for the proposed R-Q model

For the implementation, the proposed algorithm consists of three stages:

1. Pre-encoding each tile, derive the model parameters;

2. Input the parameters into the final eq. 3.5 and derive the optimal QP for each tile.

3. Actual encoding based on the assigned QP.

Finally, each tile is encoded to produce the final bit stream stored at the server at the target bit rate.

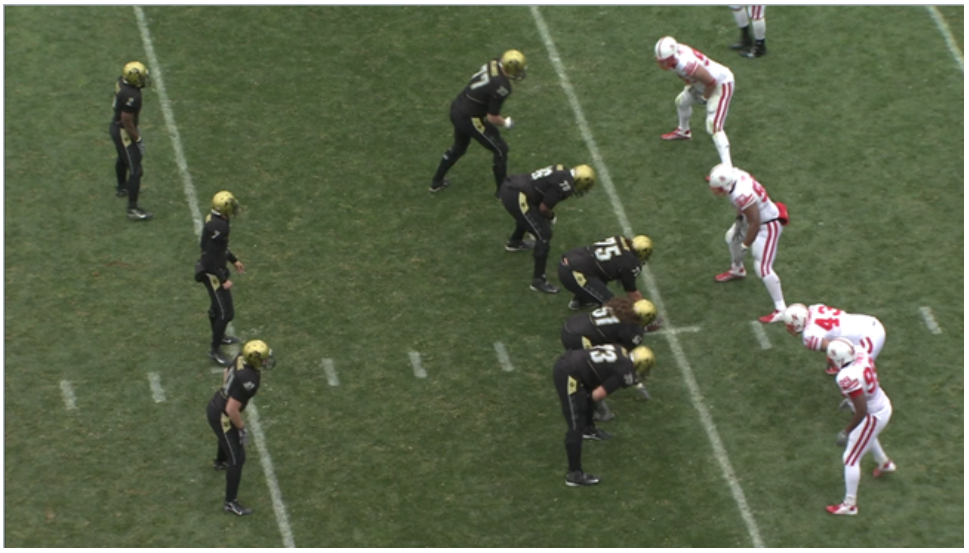
3.1.4 Experimental Result and Conclusion

To assess the techniques presented in this section, we collected two sets of data from two HD sequences, Raven and Touchdown which represents two typical sequences. Raven has more difference in content across different tiles. Touchdown has less complex content across the whole frame. Each sequence has a resolution of 1920 × 1080 pixels and at 30 frames/second. The Interactive Region of Interest (IRoI) encoder in classX creates 2 dyadic resolution layers with a total of 5 tiles. Each tile has 480 × 270 pixels and is encoded into an H.264/AVC bit-stream using x264(v.0.77) codec library. The motion estimation is set to have quarter-pixel accuracy with a search range of 16 pixels.

We consider four tiles in the second layer without losing the generality here. We can easily generalize this case to the third layer. In order to simulate the real client behaviour and test all kinds of situation, we assume four probability sets, as shown in table 3.1, to simulate the probability which users choose each tile.

From table 3.2, we know that the average PSNR gain for Raven and Touchdown are 1.02dB and 0.16dB respectively. In figure 3.4 and 3.5, we can get a more

(a)



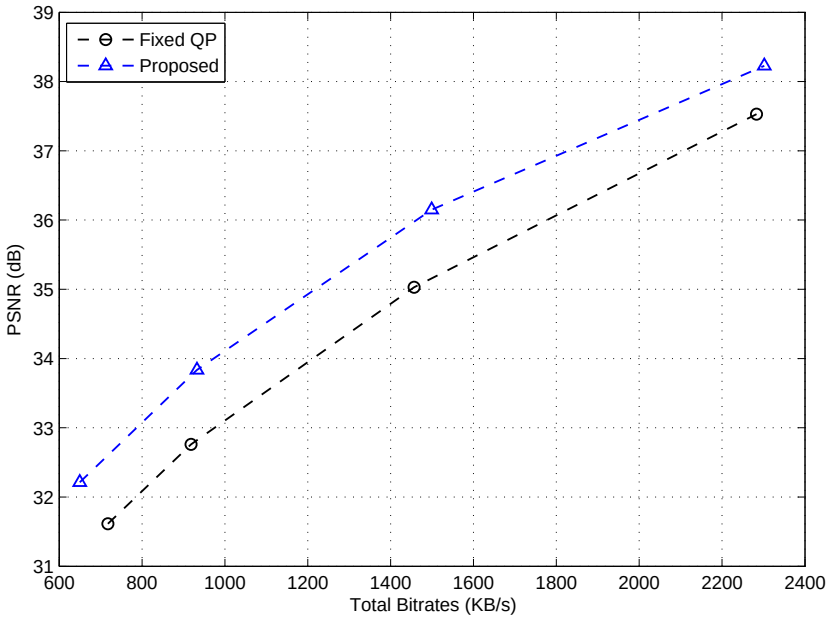
(b)

FIGURE 3.3: Content comparison between raven and touchdown

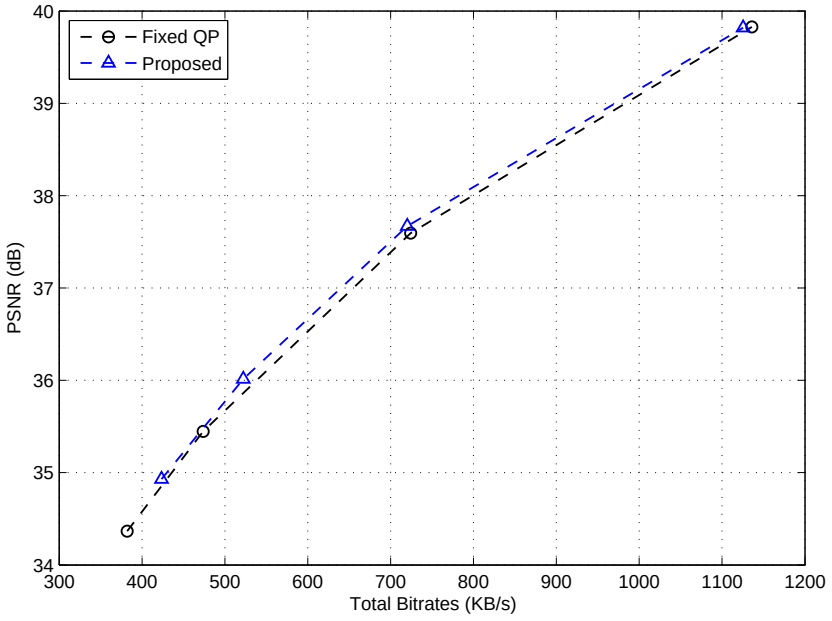
TABLE 3.1: Simulation Probability sets

	Tile1	Tile2	Tile3	Tile4
set1	0.3398	0.3175	0.1942	0.1485
set2	0.3297	0.865	0.3197	0.2641
set3	0.4026	0.1762	0.227	0.1942
set4	0.2951	0.3281	0.046	0.3308

straightforward feeling of the result. With the result above, we can make a conclusion that for a sequence whose content is diverse across different tiles,



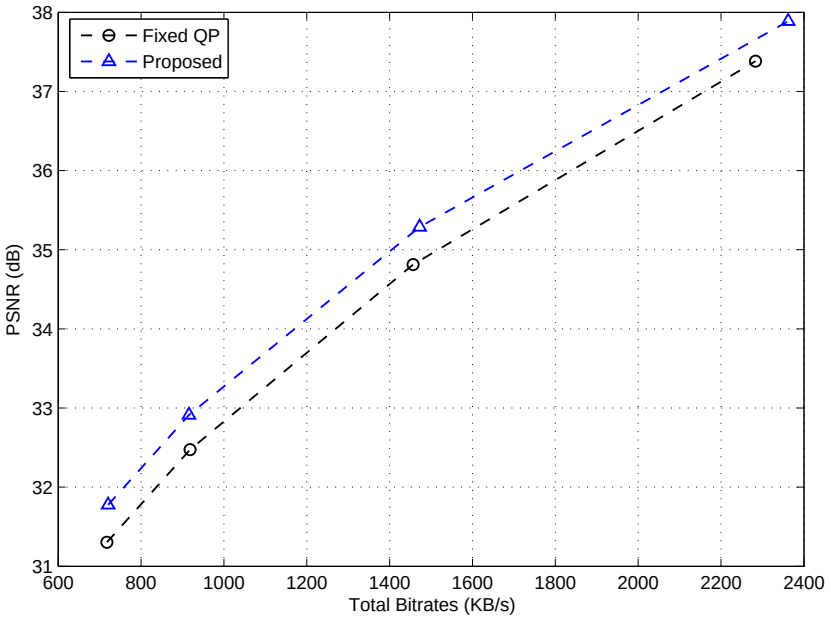
(a)



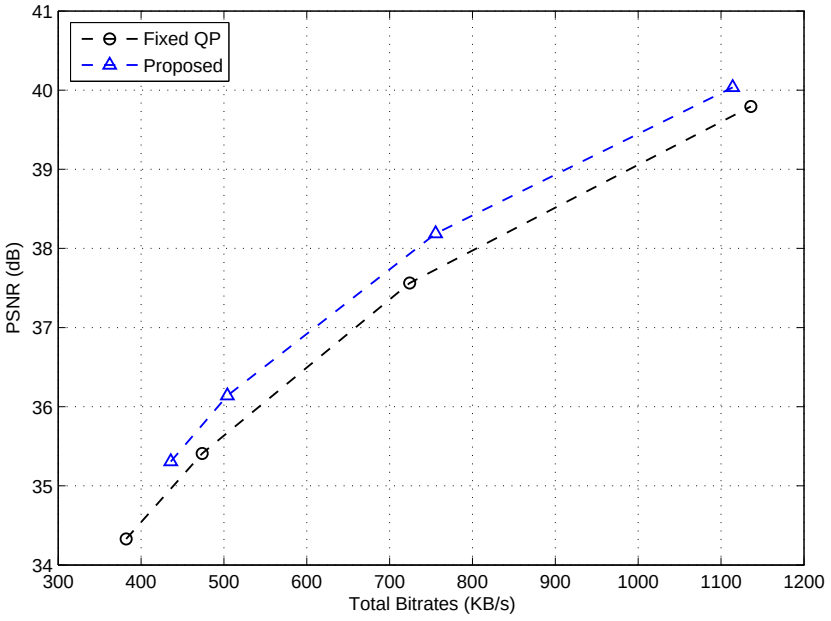
(b)

FIGURE 3.4: Optimal bit allocation in tile-based video streaming system (Probability set 1), (a)Raven (b)Touchdown

the proposed algorithm will get more gain. However, for smooth sequences, the gain is very limited.



(a)



(b)

FIGURE 3.5: Optimal bit allocation in tile-based video streaming system (Probability set 4), (a)Raven (b)Touchdown

TABLE 3.2: PSNR gain (dB) for optimal bit allocation in tile-based video streaming system

PSNR gain (dB)		
Prob Set	Raven	Touchdown
1	1.0322	0.0545
2	1.7309	0.2366
3	1.015	0.077
4	0.3149	0.2833
Average	1.02325	0.16285

3.2 A Fast Bit Allocation Algorithm for Tile-based Streaming System

In previous section, we proposed a model-based bit allocation algorithm. It can achieve up to 1.02dB PSNR gain in our test. The drawback is that we need four parameters to form the R-D models. In this section, we will present a fast bit allocation algorithm for tile-based streaming system. With this method, we can reduce the complexity without losing too much gain.

Generally, there are two ways to optimize the bit allocation in encoding. One is to keep the sum of bit rate as a constant value and maximize the subjective quality like we did in previous section. However, The parameters are calculated from pre-encoded videos which is time consuming and complex for the computer to deal with in real time. To avoid this, we take a different approach: preserve the quality while minimizing the bit rate[13]. it turns out that by doing this there is no need to use the R-Q model, thus reduce the complexity. The details of this new method are described as follows:

3.2.1 Problem Formulation

Assume that the R-D function is as follows for a given tile i[13]:

$$D_i(R_i) = \sigma_i^2 \cdot e^{-\gamma R_i} \tag{3.6}$$

in which D_i stands for the mean square error, R_i denotes the bitrate, and σ^2 is a measurement of the variance of the encoding signal and describes the complexity of the video content, γ is a constant coefficient. If we take the users' selection probability pattern into consideration, the encoding distortion in tile i can be written as follows:

$$D_i' = w_i \cdot D_i \tag{3.7}$$

here wi is the client' selection probability for each tile. Here we made an assumption for the probability distribution. In reality, the server will collect this information within a sufficient long time to get the distribution for each tile.

$$\begin{cases} min \sum_i R_i \\ s.t. \sum_i p_i D_i = D \end{cases} \tag{3.8}$$

Here R_i is the total bits used for each tile. D_i is the distortion for each tile which is encoded by H.264/AVC. p_i is the pre-defined probability assumption

for each tile. D is the target distortion. With the Lagrangian multiplier method we can solve this equation in close form:

$$\begin{cases} J(D_1, D_2, ...D_N) = \sum_i R_i + \lambda(\sum_i p_i D_i - D) \\ R_i = \frac{1}{\gamma}(\log \sigma^2 - \log D_i) \end{cases} \tag{3.9}$$

in which N is the number of tiles in the encoded image. To obtain the solution, we take the partial derivative of the the objective function:

$$\frac{\partial J}{\partial D_1} = \frac{\partial J}{\partial D_2} = ... \frac{\partial J}{\partial D_N} = 0 \tag{3.10}$$

Solving these equation above, we obtain:

$$D_i = \frac{1}{p_i} \times \frac{1}{N} \times D \tag{3.11}$$

In paper [13], they assumed that D is linear to Q.

$$D = k \times Q_{step} \tag{3.12}$$

By this relationship, they got

$$Q_{istep} = \frac{1}{p_i} \times \frac{1}{N} \times Q_{stepbaseline} \tag{3.13}$$

In order to improve the performance, we made a slight modification to this reasoning process: We use a more accurate D-Q model as follows:

$$D(Q) = e \cdot Q^{\beta} \tag{3.14}$$

From equation 3.10, we can get,

$$D_i = \frac{1}{\gamma \lambda p_i} \tag{3.15}$$

In addition to this equation, we have a relationship between D_i and D,

$$\sum_i p_i D_i = D \tag{3.16}$$

By solving the two equations above, we can get:

$$Q = \sqrt[\beta]{\frac{1}{e_i p_i DN}} \tag{3.17}$$

The advantages for this problem formulation is that we do not need to know the relationship between R and Q. We only need two parameters for each tile here. Besides, we have a closed-form solution instead of numerical calculation. The complexity has been reduced significantly compared to previous model-based algorithm.

We can make a conclusion as follows:

- Easy to implement because the final solution is a close-form solution.
- Reduce the number of model parameters we need to derive, thus decrease the complexity.

TABLE 3.3: Fast Algorithm PSNR gain in under 4 probability assumptions

PSNR Gain (dB)				
	Fast		Model-based	
Prob Set	Raven	Touchdown	Raven	Touchdown
1	0.0557	-0.0375	1.0322	0.0545
2	1.2412	0.2051	1.7309	0.2366
3	0.7429	-0.0828	1.015	0.077
4	-0.3717	0.2164	0.3149	0.2833
Average	0.417025	0.0753	1.02325	0.16285

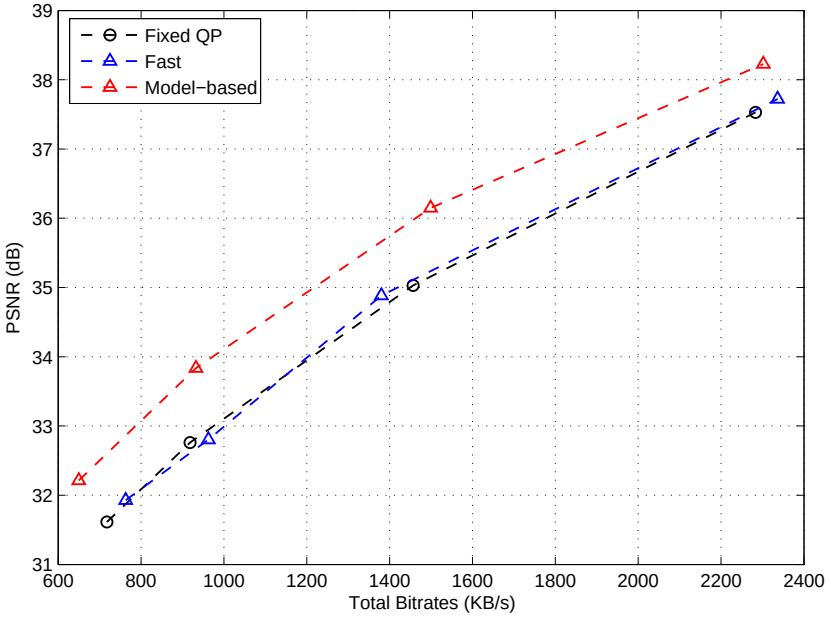
- The performance is decent.

According to the analysis above, we can apply this algorithm to guide the quantization parameter adjustment to conduct the optimized bit allocation.

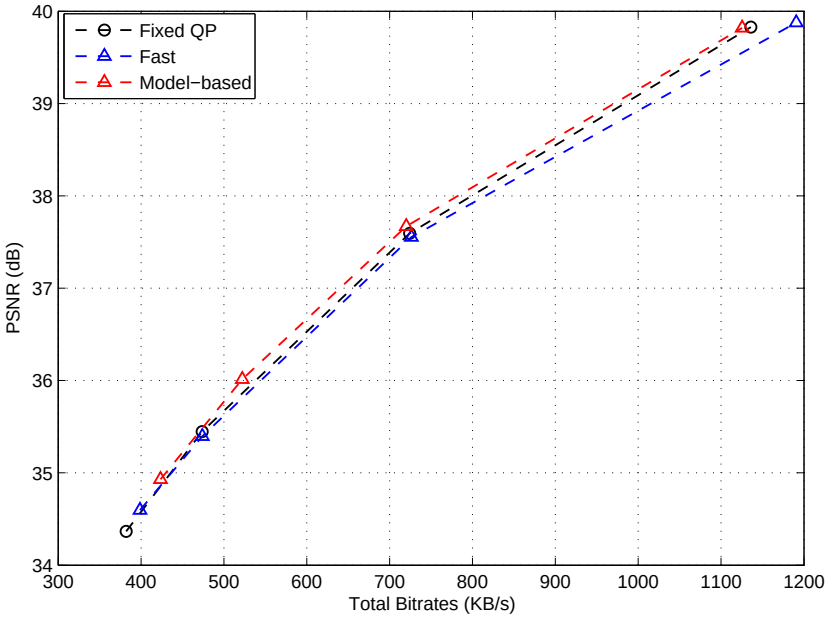
3.2.2 Experimental Result and Conclusion

The test environment setting is the same as previous section 3.1. We still use two sequences Raven and Touchdown for comparison purpose.

From table 3.3, we know that the PSNR gain for Raven and Touchdown are 0.42 and 0.08 respectively. Compared to the model-based algorithm in previous section, we will lose some gain. But in terms of complexity, we use a simple close-form solution to get the final QP assignments which is very practical for real-time implementation. The comparison is shown in figure 3.6 and 3.7.

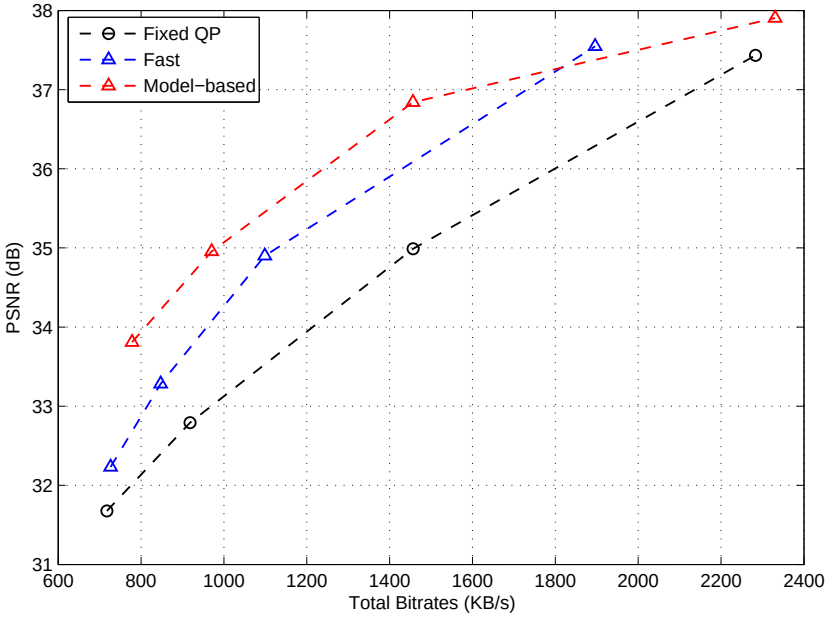


(a)

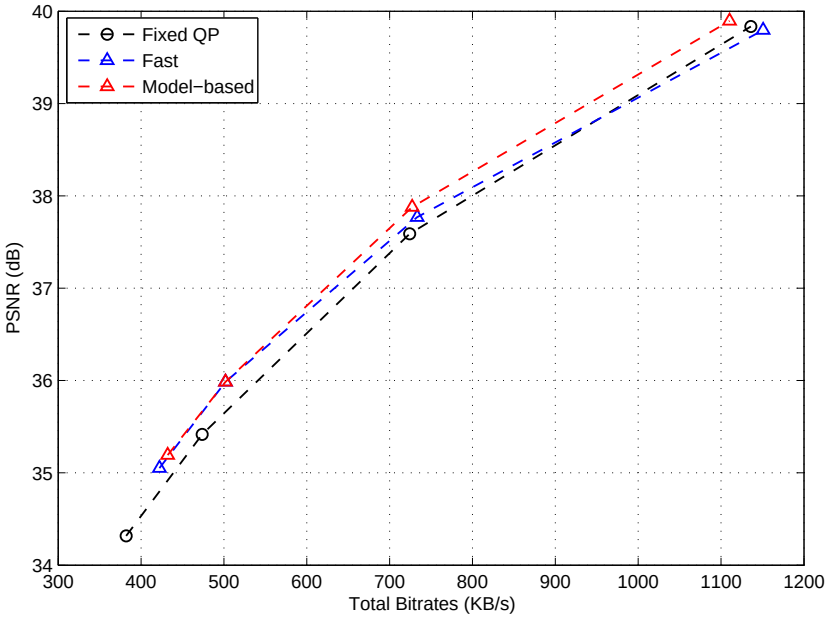


(b)

FIGURE 3.6: Fast bit allocation algorithm in tile-based video streaming system(Probability set 1) (a)Raven (b)Touchdown



(a)



(b)

FIGURE 3.7: Fast bit allocation algorithm in tile-based video streaming system(Probability set 2) (a)Raven (b)Touchdown

3.3 Optimal Bit allocation for ClassX in Transmission Scenario

3.3.1 Transmission Scenario

In this section, we consider the expected transmission rate. We need to slightly modify the previous reasoning process in section 3.1. The only change is that now we use expected transmission bitrate rather than total bitrate. The motivation is that in real transmission scenario, the average bitrate is a more accurate simulation.

In our problem formulation, we do not consider the transmission error in this thesis. The new problem can be formulated as follows:

$$\begin{aligned} &Q^* = (Q_1^*, ..., Q_N^*) = \arg\min_{Q_k \in Q} \sum_{k=1}^{N} w_k \cdot D_k(Q_k) \\ &s.t. \sum_{k=1}^{N} w_k R_k(Q_k) \leq R_{total} \end{aligned} \tag{3.18}$$

To derive the optimal solution of the Lagrangian cost function, we take the partial derivatives with respect to Q_k and λ, which yields the following equations:

$$\begin{aligned} &b\beta_k Q_k^{\beta_k - 1} + \lambda \cdot (-\alpha_k \cdot a_k \cdot Q_k^{-\alpha_k - 1}) = 0, k = 1..N, \\ &\sum_{k=1}^{N} w_k a_k \cdot Q_k^{-\alpha_k} - R_{total} = 0 \end{aligned} \tag{3.19}$$

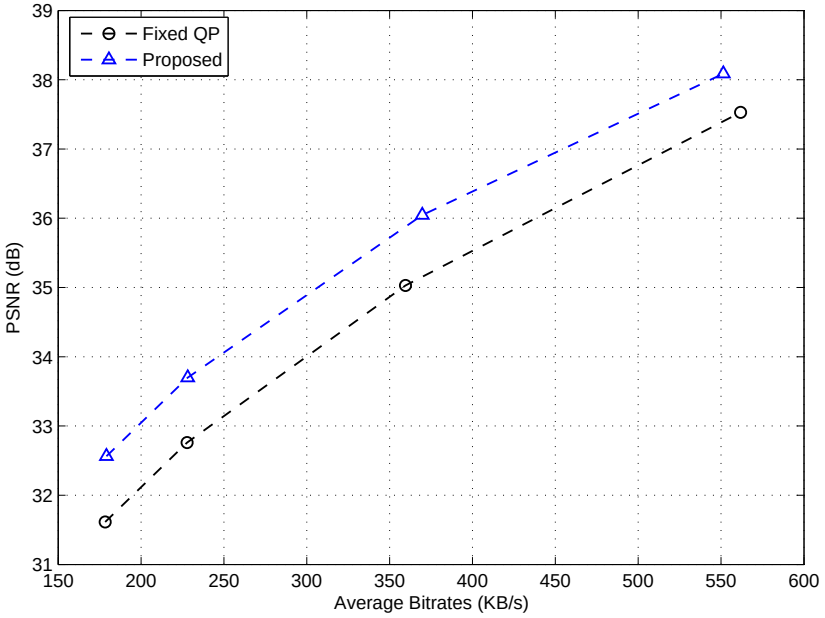
The rest steps are the same as the section 3.1.

Table 3.4: PSNR gain (dB) in transmission scenario

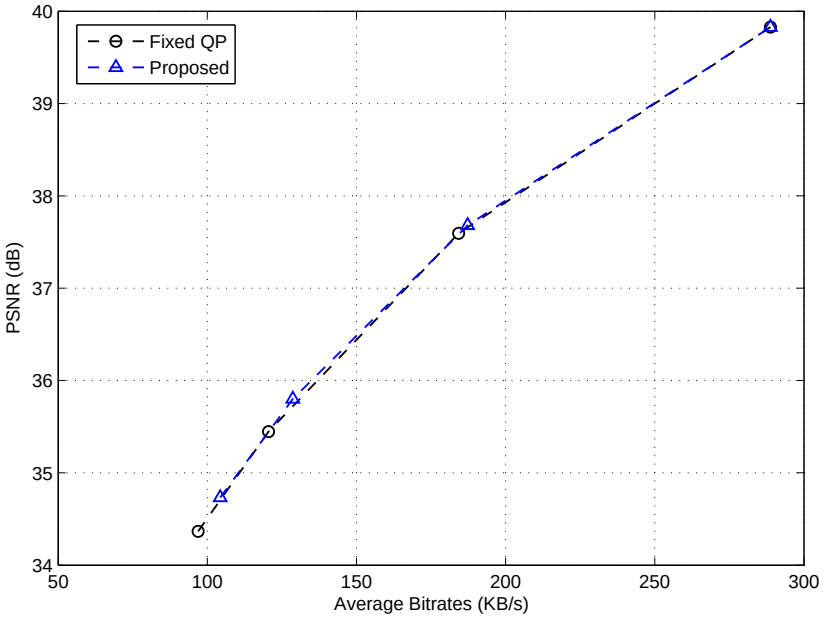
PSNR gain (dB)		
Prob Set	Raven	Touchdown
1	0.8112	0.0081
2	0.9629	0.0113
3	0.9238	0.0118
4	0.3869	0.006
Average	0.7712	0.0093

3.3.2 Experimental Result and Conclusion

The test environment and setup is the same as previous section 3.1. For sequence raven, we can get 0.8 dB gain. However, for sequence touchdown, the gain is very limited. The reason is that in transmission scenario, it is much more sensitive to the content across different tiles. We can conclude that this strategy can be only applied to videos with various contents across different regions.

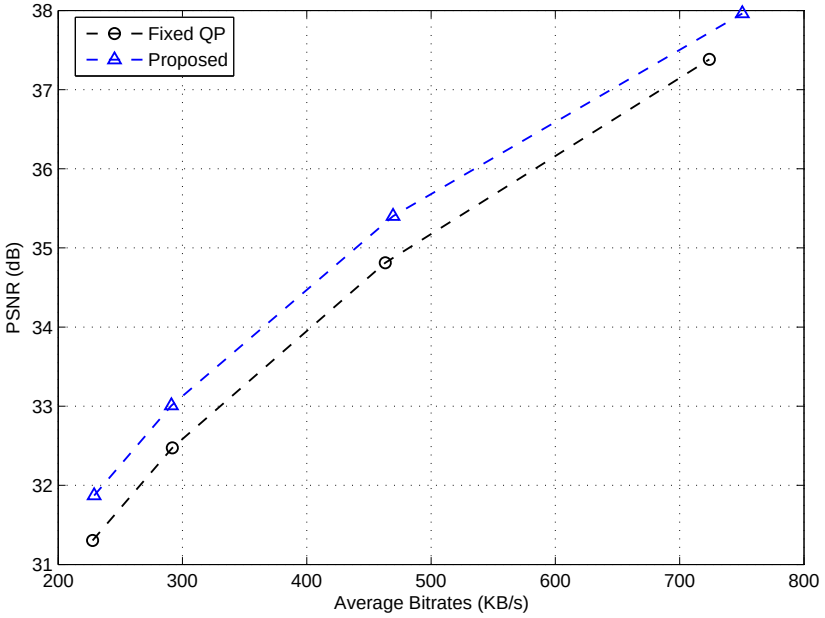


(a)

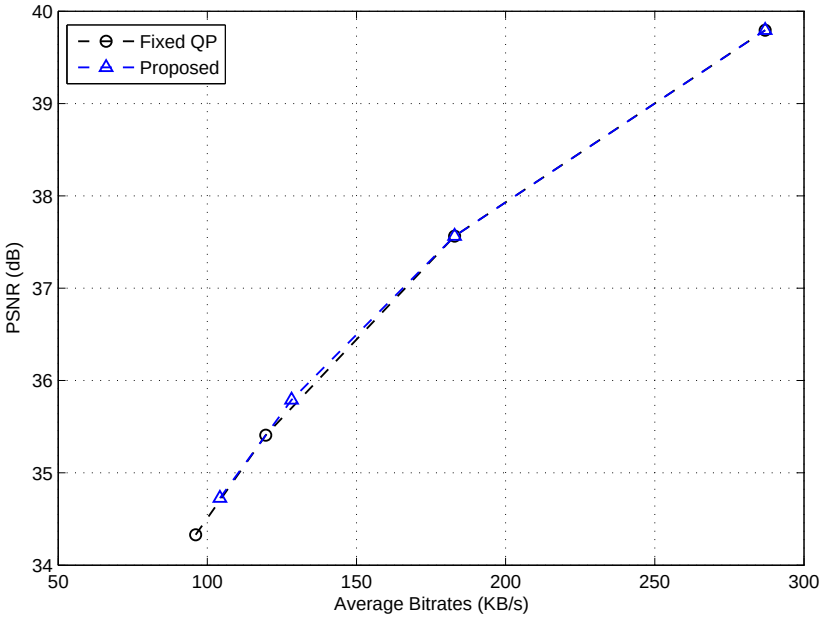


(b)

FIGURE 3.8: R-D curves in transmission scenario (Probability set 1) (a) Raven and (b)touchdown



(a)



(b)

FIGURE 3.9: R-D curves in transmission scenario (Probability set 4)(a) Raven and (b)touchdown

Chapter 4

SHVC Implementation for ClassX

4.1 Background For HEVC and SHVC

Nowadays, the display size for electronic devices have a large range, from small smart phones such as iPhone to tablets and even larger PC screen and digital TV. More and more Apps such as image sharing and video conference, require video streaming system that deal with the transmission under the conditions that display resolutions, computing abilities are different. The problem here is how to meet these various requirements efficiently. In these circumstances, scalable video coding (SVC) shows an promising potentials to solve this problem[14][15]. In SVC, we only need to encode the video sequence once, we can create a stream with spatial, quality and temporal scalability. In another words, it enables the decoder to get the video according to the specific display

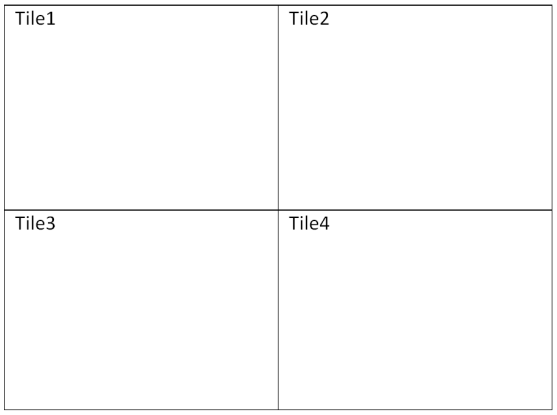


FIGURE 4.1: Tile partitioning example in HEVC

size and rate. Compared to traditional non-scalable coding, scalable video coding has the ability to save more bandwidth. Therefore, it will provides a solution for large-scale streaming system and improve the user experience.

SVC has a long history and it has been studied widely. Most previous video coding standards, such as MPEG4, H.264/AVC, have extensions to support scalability features. However, the weakness for SVC is that it increases the decoder complexity which is critical to mobile platform. These problems limits the popularity in market. Currently, a scalable extension for the new video coding standard High Efficiency Video Coding (HEVC) is under development to solve the problems above.

HEVC is the latest video compression format, a successor to H.264/AVC (Advanced Video Coding). The most impressive improvement for HEVC is that it nearly double the data compression ratio compared to H.264/AVC at the same level of video quality. It can support 8K Ultra HD and resolutions up to 8192 x 4320 which is very promising to be the next generation TV resolution. Besides, HEVC adopted a number of advances in video coding technology. The first version of the standard was completed and published in early 2013[16].

One significant feature in HEVC is tile: The operation to partition a picture into rectangular regions. Tiles are independently decodable because they are encoded with some shared header information. The main purpose of tiles is to provide the capability for parallel processing such as multi-thread and multi-core processing. This built-in feature can be naturally used in tile-based video streaming system. There are at least two benefits: Firstly, we exploit the dependency across different resolution layers. Secondly, we replace the H.264 by HEVC which provides a significant coding gain.

4.2 Motivation and SHVC Implementation in ClassX

In ClassX, in case a given tile is not available at the client end, it fills in missing pixels by upsampling relevant parts of the thumbnail(The lowest resolution has just one tile). Thus, at the same time, there are always at least two tiles being transimitted to the client. Since the content between different layers are highly correlated, without losing the generality, we can exploit the redundancy through two layers[3].

There are state-of-art techniques that enable SHVC to reduce the bitrate needed to encode spatial layers. The most important one is inter-layer prediction. In SHVC, inter-layer prediction is employed by inserting inter-layer reference (ILR)

pictures into the enhancement-layer reference picture list(s) for enhancement-layer motion-compensated prediction. In another word, the prediction of enhancement-layer is from three kinds of prediction frames, the first one is formed by motion-compensated prediction from reconstructed base-layer frame, the second one is from temporal prediction within the current enhancement-layer. The third one is from averaging base-layer reconstruction signal with a temporal prediction signal[20].

By the above inter-layer prediction in SHVC and the tile functionality naturally built in HEVC, it is motivated to use SHVC for tile-based streaming system. Thus by replacing H.264 in ClassX with SHVC, we can further reduce the bitrate for encoding the panorama sequence.

The only problem we need to solve is that: In the latest SHVC software 3.0.1, it does not support flexible tiles assignment, i.e. the number of tiles for different layers should be the same. By modifying the current software, we can set different tile numbers for different layers. In our setup, we have one tile in the base layer, and four tiles in the enhancement layer.

The functionality of assigning different tiles number in different layers is implemented based on test model SHVC 3.0.1. The configuration condition random access (RA) is tested for the sequence BasketballDrive. The BD-rate [18] is used as the criterion to evaluate the coding performance. And, both the bit rates of base-layer pictures and enhancement-layer pictures are considered when calculating the BD-rate. Fig. 4.2 demonstrates the result for implementing SHVC in ClassX. The simulcast means we simply use HEVC to encode the sequence

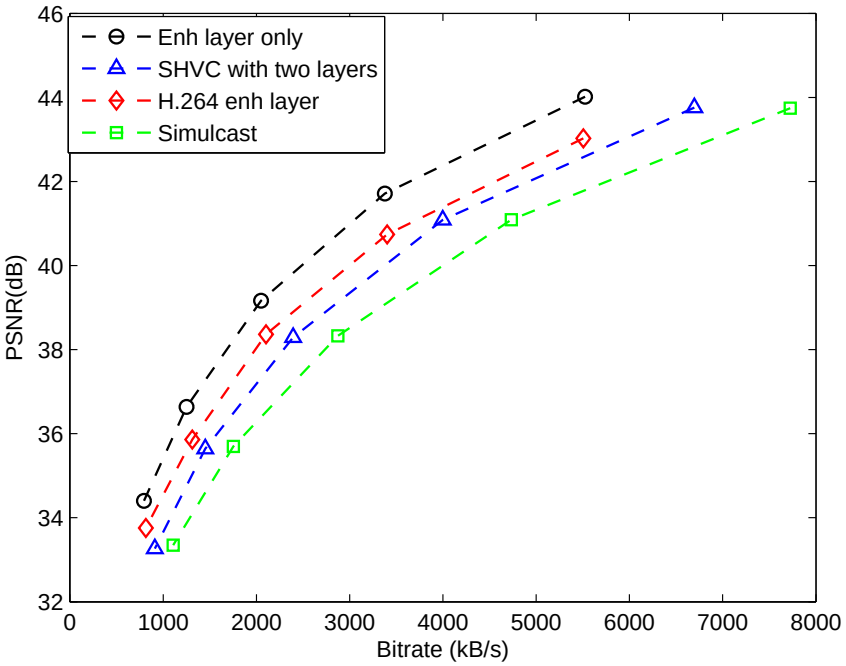


FIGURE 4.2: R-D curves for SHVC implementation in ClassX

with different resolutions, and we stream different versions with different resolutions independently without any inter-layer prediction. H.264 enh is the original ClassX coding scheme in which we only transmit H.264 stream for the enhancement layer. Enh only means that we use HEVC to encode the enhancement layer video. SHVC with two layers is the bitstream which contain two resolutions and encoded by SHVC. From this figure, we know that, compared to simulcast, we can significantly improve the R-D performance by using SHVC. In addition, compared to original ClassX coding scheme which using H.264/AVC, we can use nearly the same bitrate for the enhancement layer to encode our two layers bitstream by using SHVC.

4.3 Joint QP Optimization for Spatial Scalability in SHVC

Since SHVC is still under development, no inter layer QP optimization scheme is proposed right now. Based on the method in [11], we propose a model-based spatial layer bit allocation algorithm for SHVC in this thesis. The challenge of this problem lies in the fact that the rate distortion (R-D) behavior of an enhancement layer is dependent on its base layers because of inter-layer prediction. It is shown by experimental results that the proposed two-layer bit allocation algorithm can achieve a better coding performance. A group of pictures-based spatial layer bit allocation schemes is implemented for simulation purpose.

4.3.1 Problem Formulation

In spatial scalability of SHVC, a video is encoded in such a way that the output bit stream provides various spatial resolutions. Assume that the bit budget for one GOP of spatial layers in H.264/SVC is given. Within one GOP, the objective is to allocate the bit budget to different spatial layers of the whole GOP, and then assign the bit budget to each tile within the same spatial layer.

Note that a practical rate control algorithm for SHVC spatial scalability in tile-based streaming system operates at two levels. 1) the layer-level bit allocation by selecting a quantization parameter for each spatial layer for one GOP. In this level, we get a target QP for the enhancement layer to guide us select the QP for each tile in the enhancement layer; 2) the tile-level bit allocation by selecting a quantization parameter for each tile within the same spatial layer.

This section has focused on the solution to the first level. As to the second-level, we use the same QP assignment mechanism in Chapter 3 to select the quantization parameter for each tile.

The optimization problem for dependent bit allocation can be stated as follows. We seek the quantization step-size of each spatial layer so that the total distortion is minimized subject to the total bit budget constraint. For the enhancement layer, the quantization parameter is our target average quantization step-size. For base layer, since only one tile in this layer, the solution for the base layer will be the final one. Let N be the number of spatial layers in a frame. $R_k(Q_1, ..., Q_k)$ and $D_k(Q_1, ..., Q_k)$ are the rate and the distortion model of the kth layer with respect to the vector of quantization step-sizes, denotes by $(Q_1, ..., Q_k)$. Given the bit budget R_T of one GOP and two spatial layers, the bit allocation problem can be formulated mathematically as

$$\begin{aligned} \boldsymbol{Q}^* = (Q_1^*, ..., Q_N^*) = \arg\min_{Q_k \in \mathbb{Q}} \sum_{k=1}^{N} w_k \cdot D_k(Q_1, ..., Q_k) \\ s.t \sum_{k=1}^{N} R_k(Q_1, ..., Q_k) \leq R_{total}, \sum_{k=1}^{N} w_k = 1 \end{aligned} \tag{4.1}$$

where $\boldsymbol{Q}^* = (Q_1^*, ..., Q_N^*)$ is the optimal quantization parameters, and $\mathbb{Q}$ is all possible quantization parameter candidates. $Q_1, ..., Q_{k-1}$ in the R-D functions of the kth layer is dependent upon previously coded (k-1) layers. Furthermore, w_k is a weighting factor that indicates the importance of the kth layer. Thus, the total distortion is defined as a weighted sum of the distortion of each individual layer.

The Lagrangian multiplier method again can be used to convert the constrained optimization problem in Eq. 4.1 to an equivalent unconstrained optimization problem by introducing the Lagrangian cost function as

$$\begin{aligned} \boldsymbol{Q}^* &= \arg\min_{Q_k \in \mathbb{Q}} J(\boldsymbol{Q}, \lambda) \\ J(\boldsymbol{Q}, \lambda) &= \sum_{k=1}^{N} w_k \cdot D_k(Q_1, ..., Q_k) + \lambda \cdot (\sum_{k=1}^{N} R_k(Q_1, ..., Q_k) - R_T) \end{aligned} \tag{4.2}$$

where λ is the Lagrangian multiplier. To solve the problem given in Eq. 4.2, we will model the R-D characteristics of dependent layers as elaborated in section 4.3.3.

4.3.2 Bit Allocation Analysis for Two Spatial Layers

In this section, we consider the bit allocation problem for two-layer case (i.e., N=2). The solution can be generalized to multilayer case. Mathematically, the Lagrangian cost function of two-layer case can be expressed as

$$J(\boldsymbol{Q}, \lambda) = w_1 \cdot D_1(Q_1) + w_2 \cdot D_2(Q_1, Q_2) + \lambda \cdot (R_1(Q_1) + R_2(Q_1, Q_2) - R_T). \tag{4.3}$$

In the following discussion, we assume the equal importance of these two layers; namely, $w_1 = w_2 = 0.5$. This assumption makes sense when two layer are treated equally important. In order to ease the computational burden and avoid the requirement to collect all the R-D data while keeping decent optimality, we

will adopt a model-based approach that analyzes the R-D dependence between these two layers of SHVC in the next section.

4.3.3 Rate and Distortion Modeling

Generally speaking, the R-D model of a dependent layer [i.e., $R_2(Q_1, Q_2)$ and $D_2(Q_1, Q_2)$] can be represented by a function of the quantization step-size of the reference layer (Q_1) and the dependent layer (Q_2). For dependent R-D modeling, if we can convert the multi-variable rate and distortion models into several independent single-variable functions, the solution to the bit allocation problem would be significantly simplified. We will propose a way to achieve this goal in the following discussion. The following dependent model can satisfy our ultimate objective[11]:

$$D_2(Q_1, Q_2) \approx (\zeta Q_1 + \nu) \cdot Q_2^{\beta} \tag{4.4}$$

where ζ, ν and β are model parameters, which are independent of Q_1 and Q_2

$$R_2(Q_1, Q_2) = \begin{cases} r \cdot R_1(Q_1) + (s - r) \cdot R_1(Q_2/2) & \text{if} \quad Q_2 \leq 2Q_1 \\ s \cdot R_1(Q_2/2) & \text{if} \quad Q_2 > 2Q_1 \end{cases} \tag{4.5}$$

The Cauchy-density-based R-D model has a good balance between complexity and estimation accuracy. Thus, for $R_1(Q_1)$ and $D_1(Q_1)$ of the base layer, we adopt the following models:

$$D_1(Q_1) = b \cdot Q_1^{\beta} \text{ and } R_1(Q_1) = a \cdot Q_1^{-\alpha} \tag{4.6}$$

where a,b,α, and β are model parameters. With the enhancement layer R-D model given in Eq. 4.4 and Eq. 4.5, and the base layer R-D models in Eq. 4.6, we are ready to solve the bit allocation problem for SHVC with two spatial layers. Since the proposed R-D models are defined by completely closed-form expressions, a numerical solution by computing to the Lagrange formulation in 4.3 becomes viable. After pre-encoding the sequence by SHVC, we can get the rate and distortion data. With this information, we can derive the model parameters.

4.3.4 Solution to the Lagrangian Formulation

The rate model in Eq. 4.5 can be further simplified when we impose the following constraint:

$$0 \leq QP_2 - QP_1 \leq 6 \tag{4.7}$$

This constraint is usually met by the optimal solution (Q_1^*, Q_2^*). Under the above conditions, the Lagrangian cost function in Eq. 4.3 can be written as

$$\begin{aligned} J(\boldsymbol{Q}, \lambda) = \frac{1}{2}(b \cdot Q_1^{\beta_1} + (\zeta Q_1 + \nu) \cdot Q_2^{\beta_2}) \\ +\lambda \cdot ((1+r) \cdot aQ_1^{-\alpha} + (s-r) \cdot a(Q_2/2)^{-\alpha} - R_T) \end{aligned} \tag{4.8}$$

To derive the optimal solution of the Lagrangian cost function, we take the partial derivatives with respect to Q_1, Q_2, and λ , which yields the following

three equations:

$$\begin{aligned} b\beta_1 \cdot Q_1^{\beta_1 - 1} + \zeta \cdot Q_2^{\beta_2} - a\alpha(1+r)Q_1(-\alpha - 1) \cdot \lambda = 0 \\ \nu\beta_2 \cdot Q_2^{\beta_2 - 1} - 1/2a\alpha \cdot (s-r)(Q_2/2)^{-\alpha-1} \cdot \lambda = 0 \\ a \cdot (1+r)Q_1^{-\alpha} + a \cdot (s-r)(Q_2/2)^{-\alpha} - R_T = 0 \end{aligned} \tag{4.9}$$

Note that there are three variables Q_1, Q_2, and λ in 4.9 to be solved while other parameters are determined in an earlier steps. To be more specific, the proposed algorithm consists of three stages:

1) pre-encoding the sequence by SHVC for model parameter derivation,

2) Numerical calculation using eq. 4.9 and get the step size Q assignment to different layers, in our test case, the number of layers is two.

3) With the determination from the previous step, we know the target Q for the enhancement layer. By repeating the same algorithm in chapter 3, we can further improve the R-D performance.

4) actual encoding based on computed Q_1 , Q_2 and Q for each tile in the enhancement layer that optimize the Lagrangian cost function using the one-to-one correspondence between quantization step-size Q and quantization parameter QP. Finally, each layer and each tile in one GOP unit is encoded to produce the final bit stream at the target bit rate.

Table 4.1: PNSR gain for BasketballDrive

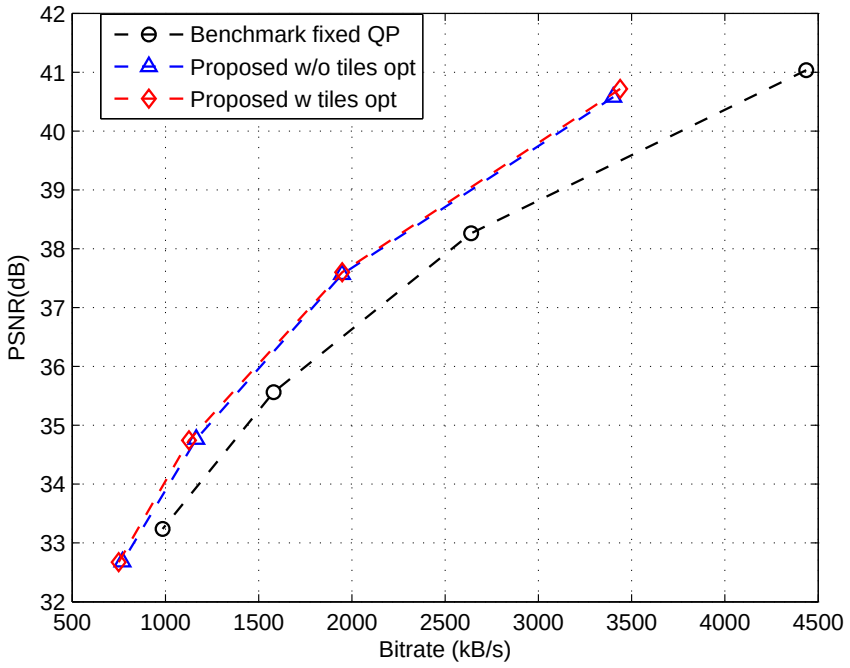
	PSNR gain after layer optimization	PNSR gain after tile optimization
Prob1	0.8284	0.1396
Prob2	0.6592	0.0011
Prob3	0.811	0.1375
Prob4	0.8799	0.1405
Ave	0.794625	0.104675

Table 4.2: PNSR gain for Kinomo

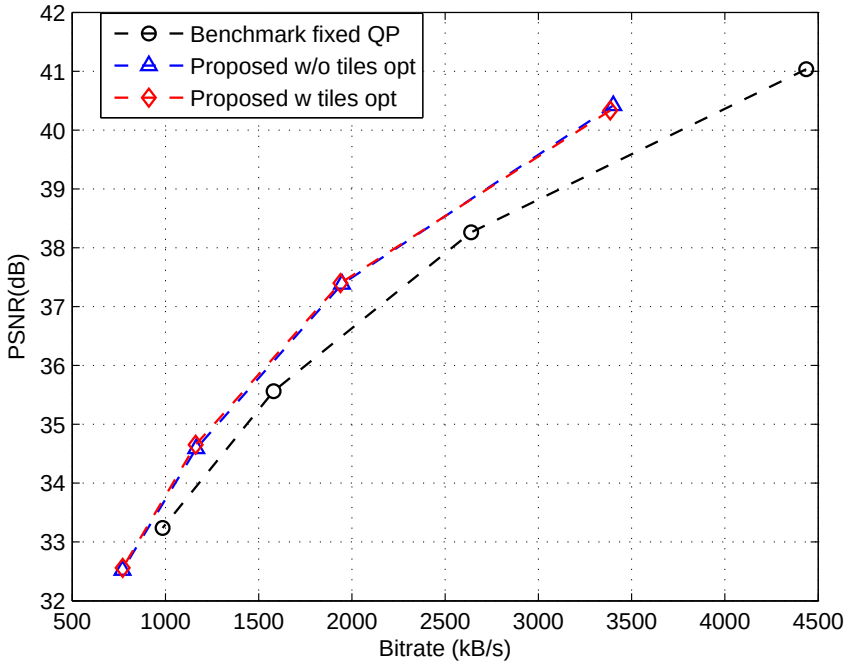
	PSNR gain after layer optimization	PNSR gain after tile optimization
Prob1	1.3008	0.2629
Prob2	1.3634	0.0773
Prob3	1.3008	0.1685
Prob4	1.3481	0.2085
Ave	1.328275	0.1793

4.3.5 Experimental Result

The proposed two-spatial-layer bit allocation algorithm was implemented based on scalable HEVC test model SHVC 3.0.1 and Matlab. Since there is no spatial layer bit allocation algorithm in the current version of SHVC software, we compare the performance of the proposed two-layer bit allocation algorithm against that of the fixed QP method[18]. We also compare the result after implementing previous tile-based QP optimization strategy in the enhancement layer, i.e. a two-step optimization scheme, the layer level optimization and the tile-level optimization. From table 4.1 and 4.2, we can figure out that the average PSNR gain among two layers for basketballdrive and kinomo are 0.79dB and 1.33dB separately. With additional optimization among the tiles in the enhancement layer, we can get another 0.1dB and 0.17dB coding gain respectively. The R-D curve comparison is showed in figure 4.3 and 4.4. The



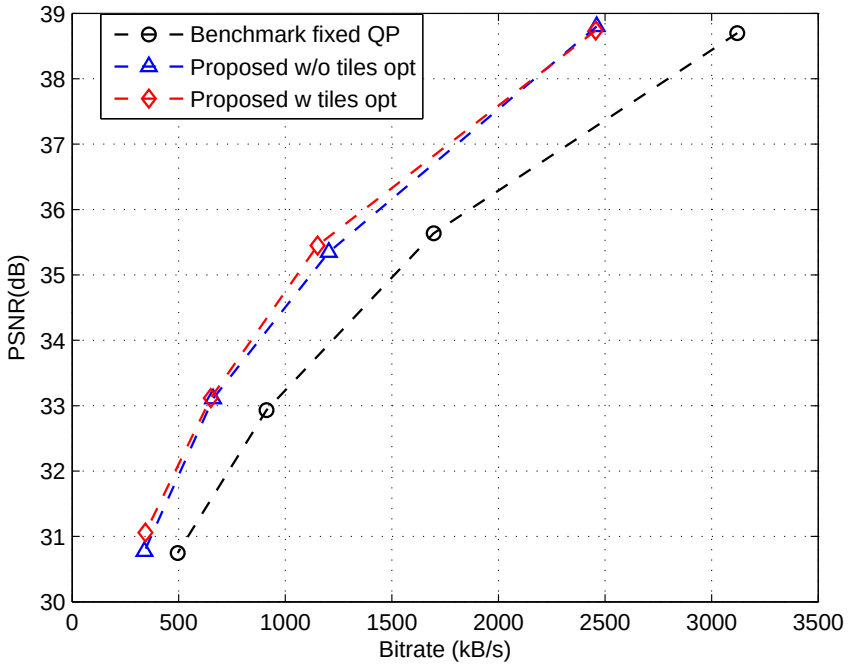
(a)



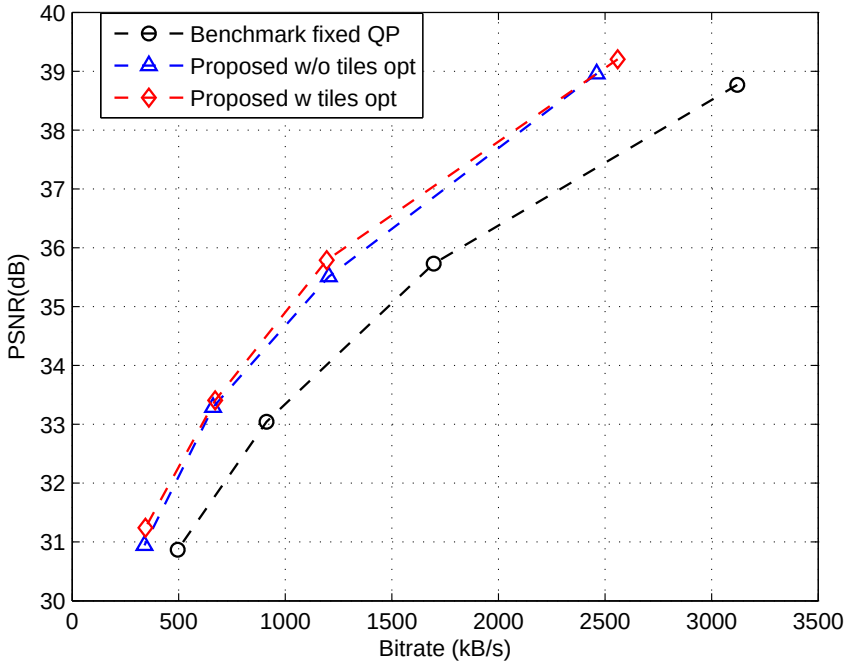
(b)

FIGURE 4.3: R-D curve comparison for BasketballDrive

provability assumption is the same as table 3.1.



(a)



(b)

FIGURE 4.4: R-D curve comparison for Kinomo

Chapter 5

Downsampling-Based Interactive Multiview Video Stream

Multi-view video as a new way to represent the 3D world has attracted massive attention due to greatly enhanced stereoscopic viewing experience. Users can choose an arbitrate view point to enjoy 3D movies, TV shows, sports game, and etc. Unlike conventional single-view video systems, a multi-view video system allows the user to choose their own view to enjoy a video, thus interactivity is the most attractive feature for this kind of application. It brings a freedom and entertainment to the user. However, this interactive multi-view video streaming (IMVS) system dramatically increases the data amount and computation burden.

In free viewpoint TV, if the selected view is not encoded, view synthesis is used to create virtual view based on the neighboring views. To simplify view synthesis, depth info is used in some systems. Therefore the rate-distortion of IMVS also involves bit allocation between texture and depth[24].

5.1 Motivation

Previous research has shown that downsampling ahead of encoding and then upsamling after decoding can improve the rate-distortion performance, especially at low bitrate[22]. Inspired by those research, this thesis proposed a downsampling-based IMVS. In Fig 5.1, we demonstrate the proposed scheme. Existing schemes use original resolution base views to synthesis virtual view. Our proposed scheme first downsamples the original base view before encoding, at the reciever side, we upsample the base view into original resolution and synthesis the virtual view.

5.2 Preliminary Result for Using Downsampled Views in IMVS

In this section, we present a preliminary result to verify that using downsampled views in IMVS can lead to improved performance at low rates. We use HEVC reference software HM 11.0 to be the encoder for base views. Since this is a preliminary result, we use independent coding strategy to simplify the test process. For synthesis part, we use HEVC 3D extension software HTM 1.0 to acquire virtual view with the help from depth view. The benchmark scheme uses the original resolutions for all encoded views. In our experiment, we encode View 1 and View 3 of the Kendo and Balloons dataset, and synthesize View 2. The original resolution is 1024 x 768.

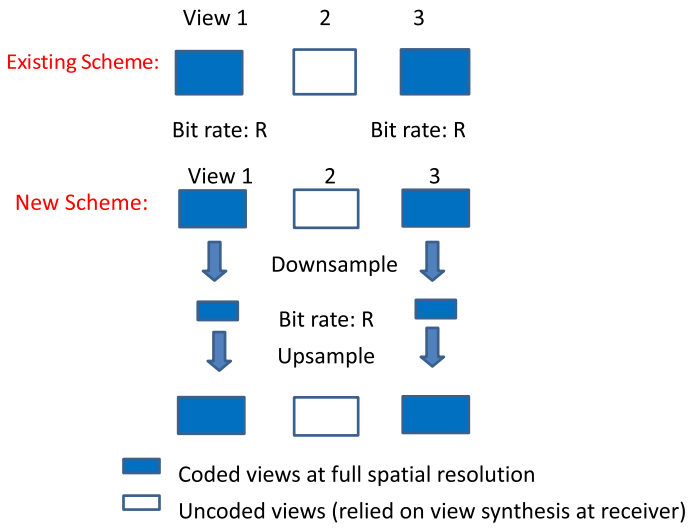


FIGURE 5.1: Proposed scheme for IMVS

The proposed scheme downsamples the coded views before encoding them, and upsamples them after decoding. The upsampled views are then used to synthesize virtual views. The downsamling ratio is 0.5 in this section. In the next section, we will discuss the bitrate adaptive downsampling ratio selection.

The detailed experimental steps are:

1. Downsample the original Views 1 and 3.

2. Encode them by HEVC, set a QP, get the bitrate for each view. Decode the sequence, and upsample to the original resolution.

3. Based on the bitrates above, find the corresponding QP for the original 1024*768 resolution sequences such that they will have the same bitrates. These are called degraded sequences here, because they will need higher QP values to get the same rates.

4. This completes the preprocessing part: we have downsampled-then-upsampled sequences, and degraded sequences with original resolution. They have the almost the same bitrates.

5. We then synthesis View 2 in the two methods, and get its PSNR.

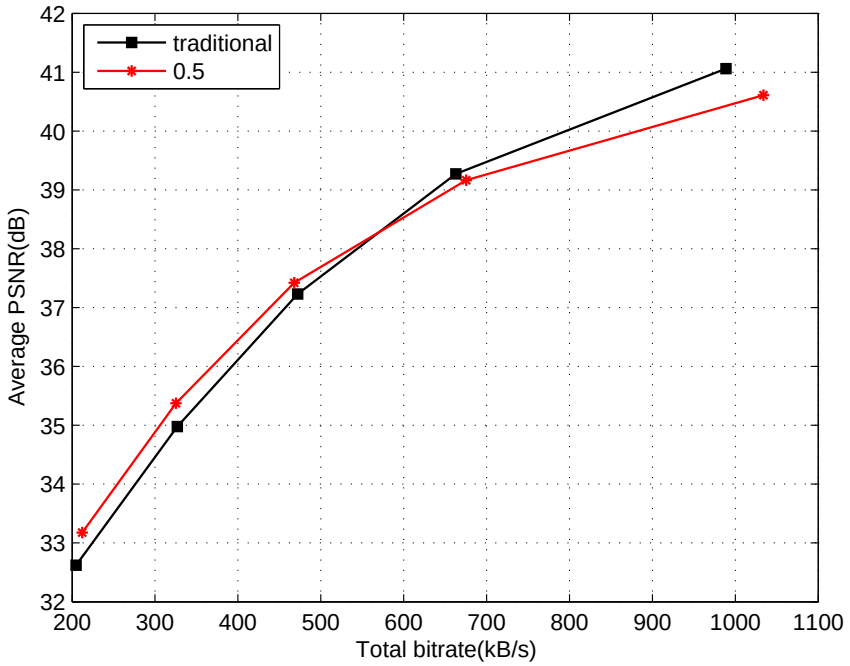
Note that for PNSR, we use the average PSNR for three views, two views and one virtual view. The bitrate is calculated by the sum of two base views.

The corresponding R-D curves are shown in Fig 5.2. As expected, the proposed method has better performance than benchmark at low rate. For sequence Kendo, the average PSNR gain is 0.18dB for all rates. If we only measure the gain below 600kb/s, the average gain will be 0.51dB. For sequence Balloon, the average PSNR gain is 0.06dB, If we only measure the gain below 700kb/s, the average gain will be 0.3dB.

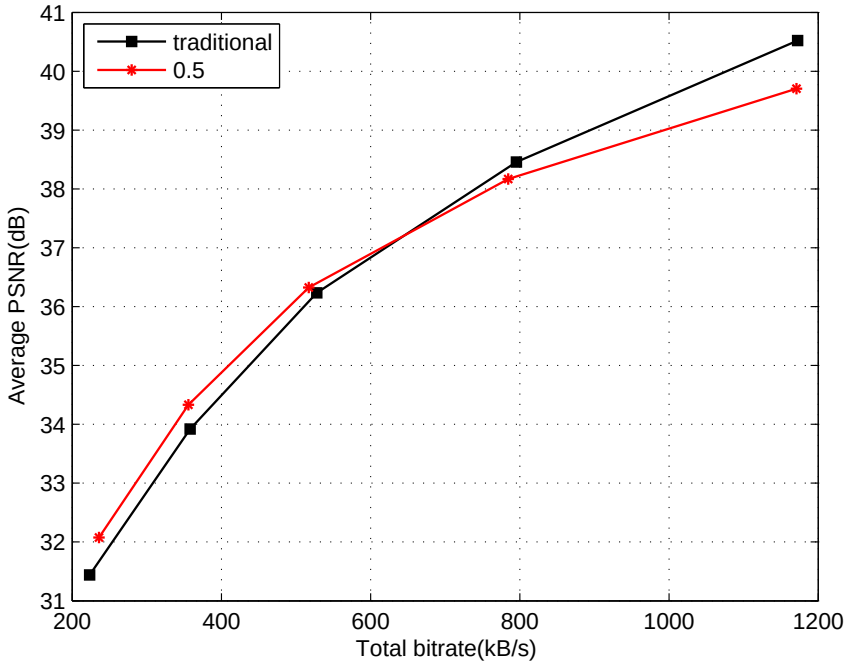
Based on these preliminary results, we can conclude that by encoding some views with reduced resolution, we can get better overall performances for both the downsampled views as well as other synthesized views at low rates.

5.3 Bitrate-adaptive Downsampling Ratio Selection

From previous section, we noticed that in high rate, the fixed downsamling ratio 0.5 will not be useful anymore. This phenomenon inspired us to create bitrate adaptive downsamling ratio selection approach to further improve the performance.



(a)



(b)

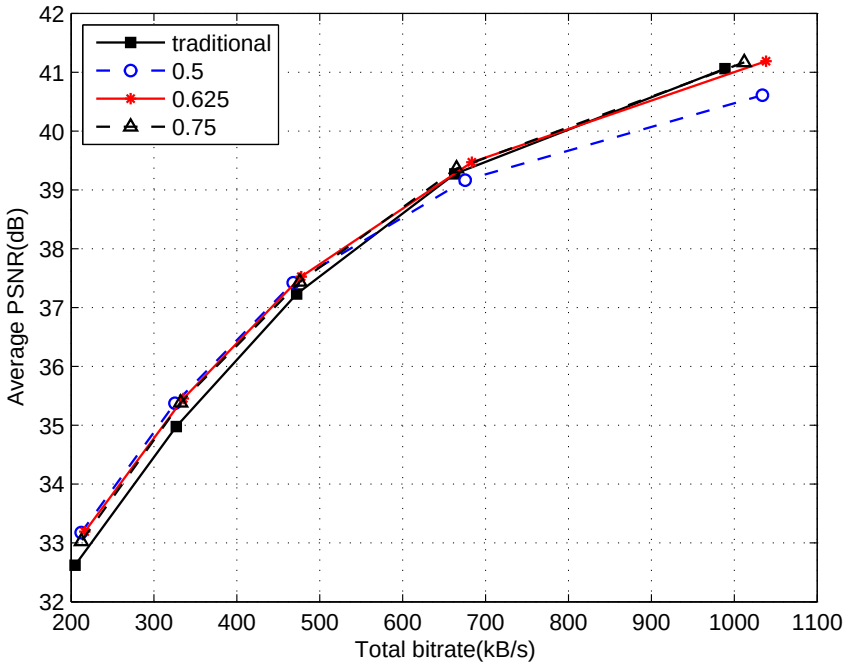
FIGURE 5.2: Preliminary result for downsampling-based IMVS, the R-D comparison for Kendo and Balloon

In this strategy, we assume three downsampling ratios, 0.5, 0.675, 0.75, the corresponding resolutions are 512*384, 640*480, and 768*576. In the first step, as shown in Fig.5.3, we simply collect the R-D points for different cases. With this information, we can pick the optimal point. For example, for sequence kendo, in the range 0-500kb, we choose 0.5, in the range 500-750kb/s, we choose 0.675. In the range 750-1000kb/s, we choose 0.75. Compared to the fixed downsampling ratio, for sequence balloons, we can get another 0.16 dB gain. For sequence kendo, this number will be 0.15 dB.

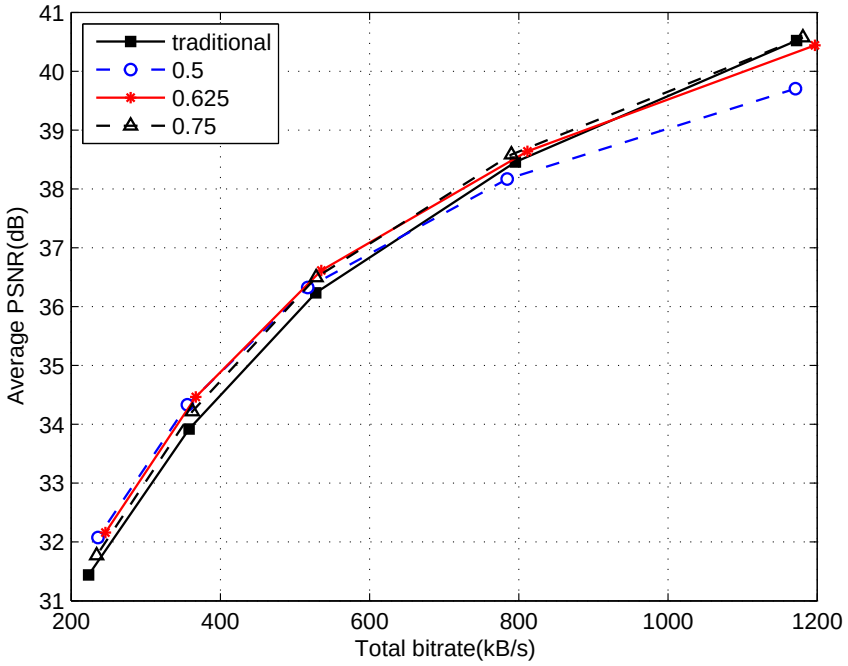
5.4 Model-based Joint Depth and Texture QP Optimization

Depth-Image Based Rendering (DIBR) view synthesis technology is used in IMVS. Depth image records the distance information between objects and the camera. Texture image along with depth image can generate the virtual view. Consequently, the compression of texture and depth images is required. Yannick [23] combined both depth and texture data simultaneously into a joint R-D surface model so that enables to find the optimal bit-allocation. Cheung[25] proposed a trellis based algorithm to joint optimize both texture and depth maps. However, there is no R-D optimization scheme for downsampled IMVS since this framework is new.

In this section, we present a model-based joint depth and texture optimization scheme. The critical point for this strategy is finding a model for the synthesised

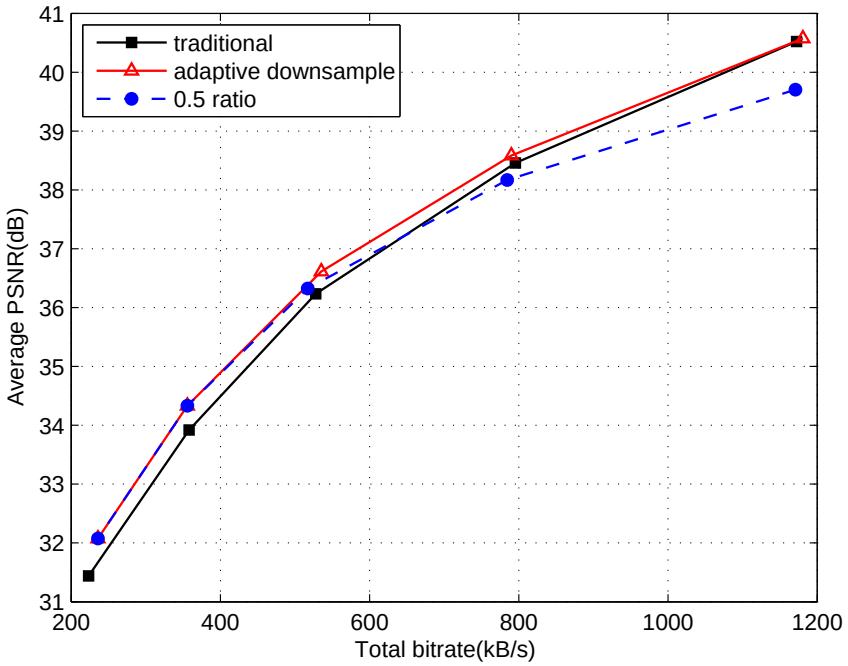


(a)

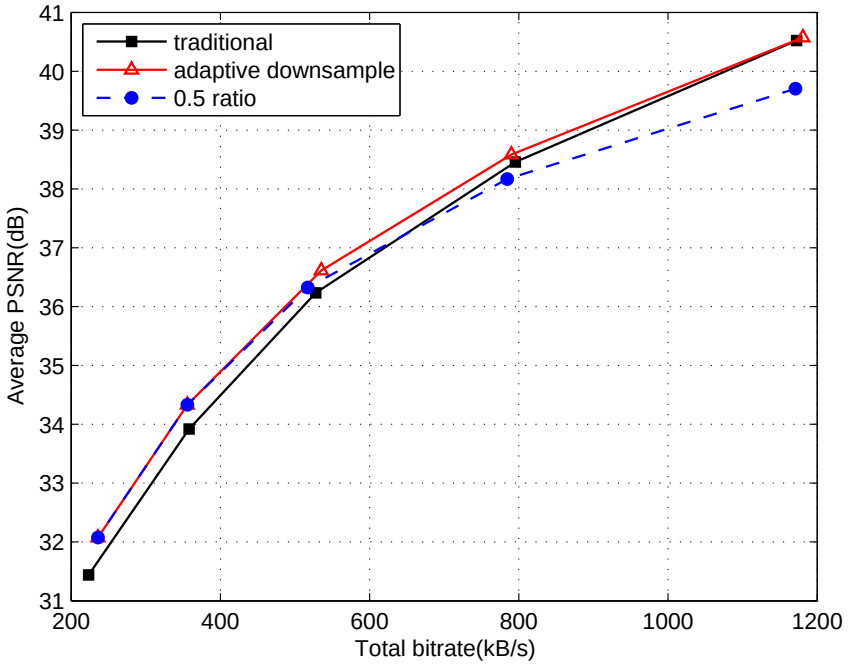


(b)

FIGURE 5.3: R-D curves for different downsamling ratio (a)Kendo (b) Balloons

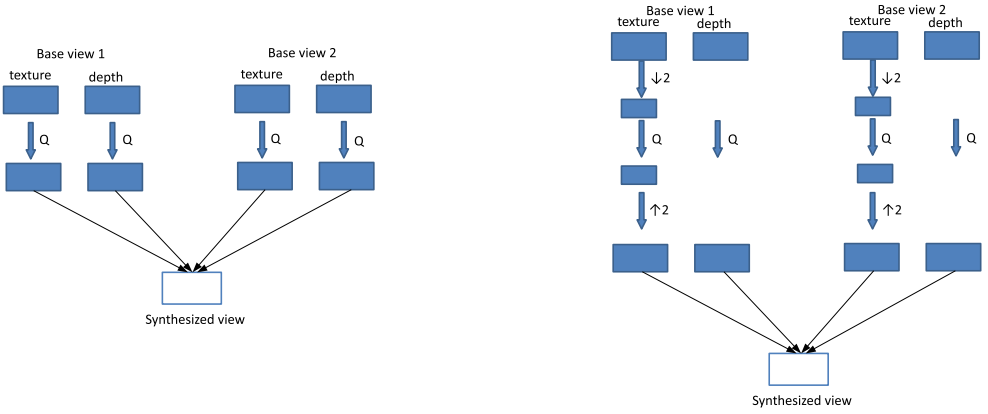


(a)



(b)

FIGURE 5.4: Rate adaptive downsampling ratio selection result (a)Kendo (b) Balloons



(a) (b)

FIGURE 5.5: Comparison between original IMVS and proposed IMVS

view. We follow the virtual view model in [21]:

$$D_v = AD_t + BD_d + C \tag{5.1}$$

D_v here is the distortion for the synthesised view. D_t is the average distortion for the texture video. D_d is the average distortion for the depth maps. A, B and C are model parameters.

Traditionally, D-Q model can be regard as a linear relationship.

$$D_t = \alpha Q_t + \beta_t, D_d = \alpha Q_d + \beta_d \tag{5.2}$$

Since the left base view and right base view nearly have the same R-D curve, so here we actually have an assumption that $Q_{tL} = Q_{tR} = Q_t$, and $Q_{dL} = Q_{dR} = Q_d$. This assumption makes sense when the base views are encoded independently. With above three equations, we can get

$$D_{total} = A_t D_t + B_t D_d + C = A_t(\alpha_t Q_t + \beta_t) + B_t(\alpha_d Q_d + \beta_d) + C = \mu Q_t + \nu Q_d + E \tag{5.3}$$

For R-Q model, we have[21]

$$R_t = a_t Q_t^{-1} + b_t, R_d = a_d Q_d^{-1} + b_d \tag{5.4}$$

Where R_t is the bitrate for texture, R_d is the bitrate for depth maps.

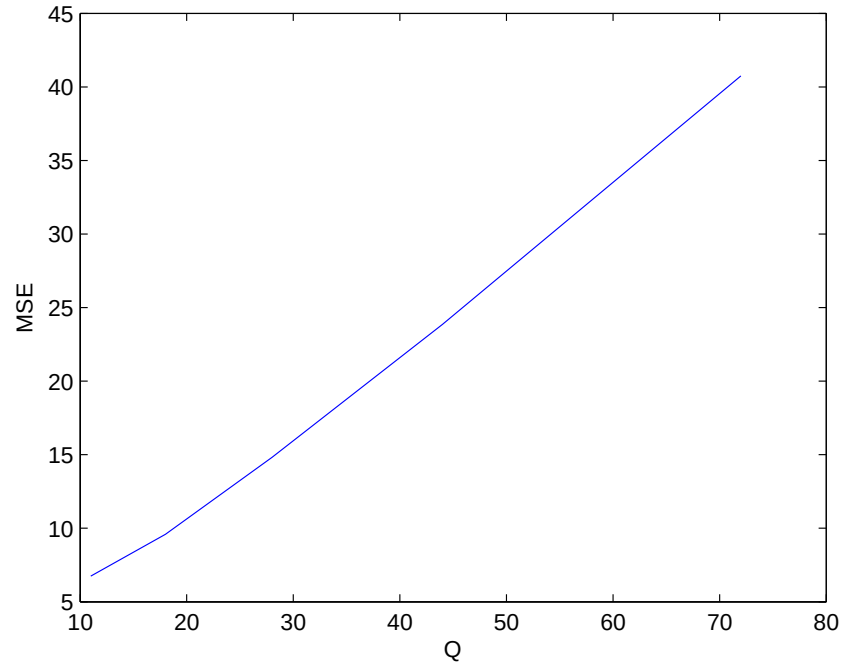
Note that even we downsample the original frame, we can still use the same form for R-D and R-Q model, which can be verified. From Fig. 5.6, the R-D and R-Q model is still suitable for downsampling case.

Based on above models, We can formulate the optimization problem:

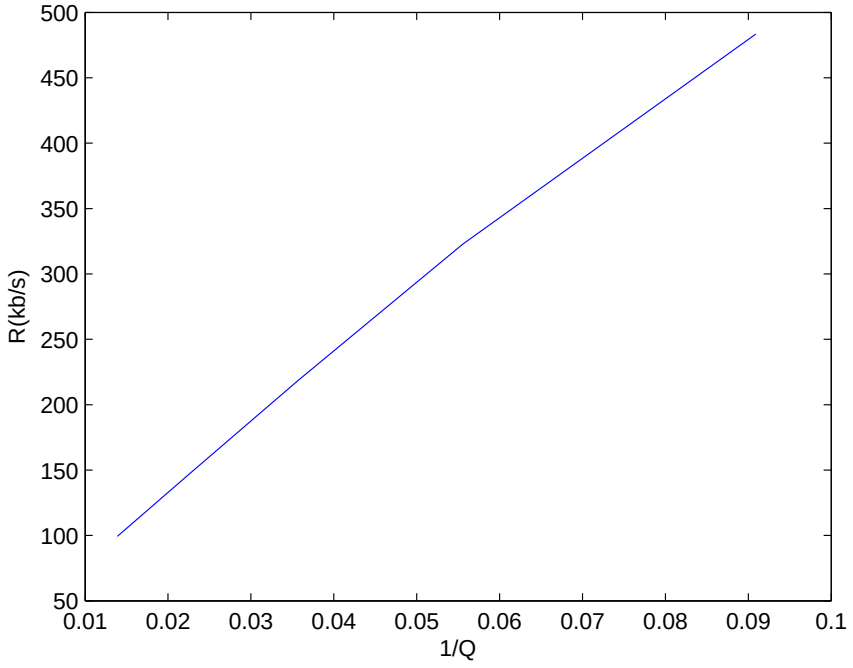
$$\begin{aligned} &\min_{Q_t, Q_d} (\mu Q_t + \nu Q_d + E) \\ &s.t.\ a_t Q_t^{-1} + b_t + a_d Q_d^{-1} + b_d \leq R_c \end{aligned} \tag{5.5}$$

With the objective function and Lagrangian multiplier λ, we have

$$\min_{Q_t, Q_d} F = \mu Q_t + \nu Q_d + E + \lambda(a_t Q_t^{-1} + b_t + a_d Q_d^{-1} + b_d - R_c) \tag{5.6}$$



(a)



(b)

FIGURE 5.6: R-D and R - 1/Q curve

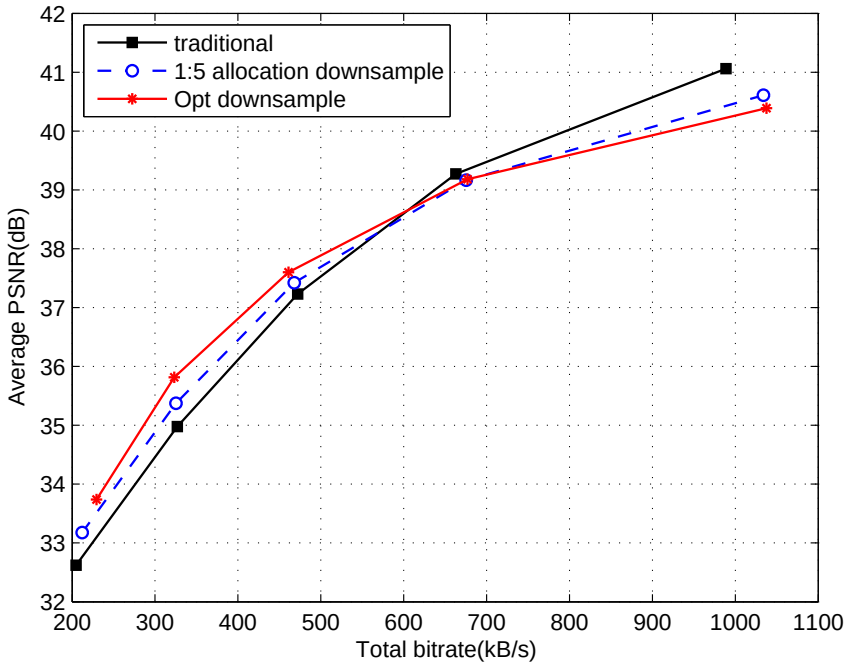
Take the partial derivative respect to Q_t, Q_d and λ, we can get:

$$\begin{aligned} \mu - \lambda a_t Q_t^{-2} &= 0 \\ \nu - \lambda a_d Q_d^{-2} &= 0 \\ a_t Q_t^{-1} + b_t + a_d Q_d^{-1} + b_d - R_c &= 0 \end{aligned} \tag{5.7}$$

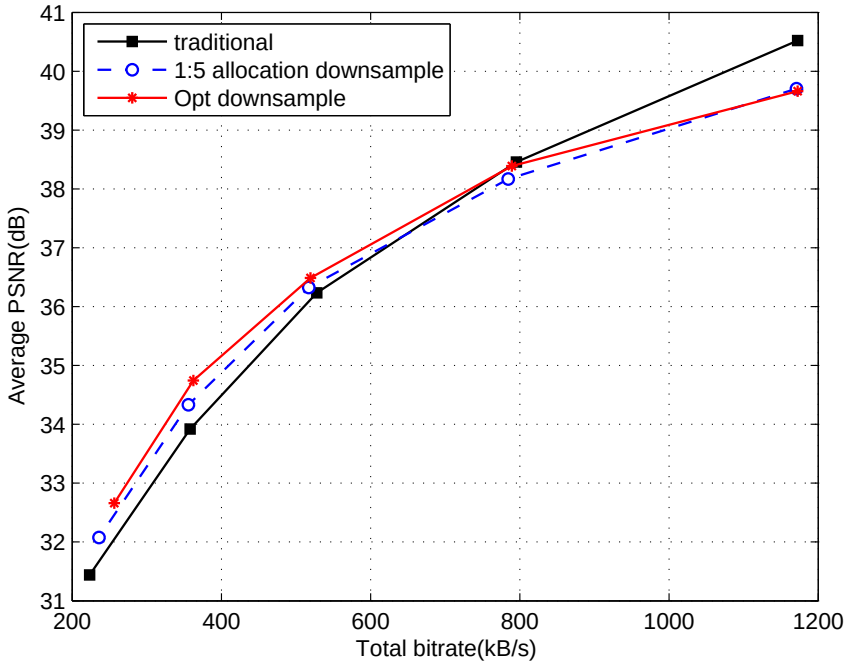
The solution is as follows:

$$Q_t = \frac{a_t + \sqrt{\frac{\nu a_t a_d}{\mu}}}{R_c - b_t - b_d},\ Q_d = \sqrt{\frac{\mu}{\nu}\frac{a_d}{a_t}} Q_t \tag{5.8}$$

With this solution, we can further improve the downsample-based IMVS. For sequence Kendo and Balloons, we can get another 0.14dB gain and 0.12 dB respectively compared to fixed 1:5 (depth:texture) bit allocation strategy for downsample-based IMVS as shown in fig.5.7.



(a)



(b)

FIGURE 5.7: Joint depth and texture QP optimization for downsampling-based IMVS (a)Kendo (b) Balloons

Chapter 6

Conclusion

In Chapter 2 and 3 of this thesis, we presented panorama video system based on tiling method and an algorithm that can optimize R-D performance across multiple tiles to eliminate extra consumption of bandwidth. The motivation behind this is not to send the whole frames or images in the 3D panorama application. According to our results, the proposed algorithm is better in some cases where the sequence has complex content for different tiles. Then, we simplify the algorithm by formulating the objective function from another perspective to reduce its computational complexity, assuming that the objective is to preserve the overall PSNR. In addition, we apply this algorithm in transmission scenario with slight modification to the objective function.

Chapter 4 presents a replacement of origial H.264/AVC coding system in ClassX. With brand new video coding standard HEVC and SHVC and our novel inter-layer bit allocation algorithm, we significantly improve the R-D performance for our tile-based panorama video system.

Chapter 5 presents a novel downsampling-based IMVS, where a full resolution video can be downsampled and quantized first, then upsampled back to original resolution to synthesis virtual views. Next, we propose a bitrate adaptive downsamling ratio selection to find the best downsampling ratio, based on the target bit rate. This approach, along with joint depth and texture optimization algorithm, allows us to improve the overall PNSR for both base views and virtual views under the same bit budget. Future work includes how to generalize the proposed optimization framework to more than 3 views and find a simplified R-D model to get an one-pass optimization algorithm.

Our technology is promising in changing the way videos displayed and shared in social network, enabling a more interactive user experience for viewing multiple panorama videos and 3D videos.

Bibliography

[1] http://instagram.com

[2] http://www.immersivemedia.com

[3] A Mavlankar, P Agrawal, D Pang, et al., An Ineractive Region-of-Interest Video Streaming System for Online Lexture Viewing, *Packet Video Workshop (PV), 2010 18th International*, 56(4):2592–2600, 2010.

[4] C Fehn, P Kauff, et al., Asymmetric coding of stereoscopic video for transmission over T-DMB, *3DTV Conference, 2007*, pp. 1-4

[5] H Brust, A Smolic, et al., Mixed Resolution Coding of Stereoscopic Video for Mobile Devices, *3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video*, 2009.

[6] http://en.wikipedia.org/wiki/OpenGL

[7] http://en.wikipedia.org/wiki/Equirectangular_projection

[8] http://en.wikipedia.org/wiki/Cube_mapping

[9] http://wiki.panotools.org/Freepv

[10] http://www.raywenderlich.com/5235/beginning-opengl-es-2-0-with-glkit-part-2

[11] J Liu, Y Cho, Z Guo, and J Kuo, Bit Allocation for Spatial Scalability Coding of H.264/SVC With Dependent Rate-Distortion Analysis, *IEEE Transctions on Circuits and Systems for Video Technology, Vol. 20, No.7, July 2010.*

[12] S Singh, RK Sharma, and MK Sharma, Weighted Bit Rate Allocation in JPEG2000 Tile Encoding, *IJCSI International Journal of Computer Issues, Vol. 8, Issue 5, No 3, September 2011.*

[13] Z Li, S Qin, and L Itti, Visual Attention Guided Bit Allocation in Video Compression, *Image and Vision Computing*,2011, 29(1): 1-14.

[14] J Nightingale, Wang Qi, C Grecos, Scalable HEVC (SHVC)-Based Video Stream Adaptation in Wireless Networks, *Personal Indoor and Mobile Radio Communications (PIMRC), 2013 IEEE 24th International Symposium on*

[15] P Helle, H Lakshman, M Siekmann, et al., A Scalable Video Coding Extension of HEVC *Data Compression Conference (DCC)*, 2013, pp.201,210, 20-22 March 2013

[16] G J Sullivan, J Ohm, W Han Member, T Wiegand, Overview of the High Efficiency Video Coding(HEVC) Standard, *IEEE Transctions on Circuits and Systems for Video Technology, Vol. 22, No. 12, December 2012.*

[17] V Seregin, P Onno. S Liu, T Lee, C Kim, H Yang, Description of tool experiment C5: Inter-layer Syntax Prediction using HEVC Base Layer, *JCTVC-K1105, Oct. 2012.*

[18] G. Bjontegaard, Calculation of Average PSNR Differences Between RD-curves, *VCEG-M33, Mar. 2001.*

[19] ISO/IEC JTC-1/SC29/WG11 w12957, Joint Call for Proposals on Scalable Video Coding Extensions of High Efficiency Video Coding (HEVC), *July 2012.*

[20] X Xiu, Y Ye, Inter-layer Motion Field Mapping for the Scalable Extension of HEVC, *IS&T/SPIE Electronic Imaging. International Society for Optics and Photonics*, 2013: 866604-866604-7.

[21] H Yuan, Y Chang, J Huo, Model-Based joint Bit Allocation between Texture Videos and Depth Maps for 3-D Video Coding, *Circuits and Systems for Video Technology, IEEE Transactions on*, 2011, 21(4): 485-497.

[22] J Dong, Y Ye, Adaptive Downsampling for High-definition Video Coding, *Image Processing (ICIP), 2012 19th IEEE International Conference*, 2925-2928

[23] Y Morvan and D Farin, *Joint Dept/Texture Bit-Allocation For Multi-View Video Compression*

[24] W Sun, L Xu, OC Au, et al., An Overview of free Viewpoint Depth-Image-Based Rendering (DIBR) *APSIPA Annual Summit and Conference* 2010.

[25] G Cheung, V Velisavljevic, Bit Allocation and Encoded View Selection for Optimal multiview Image Representation, *Multimedia Signal Processing (MMSP), 2010 IEEE International Workshop on*, 2010: 233-238.

Appendix A

Details of Cube-based Rendering of Panorama Videos Using OpenGL and iOS 5

In October 2011, Apple release iOS 5 coming with a new set of APIs that makes developing with OpenGL much easier than it used to be. The new set of APIs is called GLKit. Four main features are included:

GLKView/GLKViewController. These classes make developer use much more simple and brief code to set up a basic OpenGL ES project.

GLKEffects. These classes provide a convenient way to get some basic lighting and texturing working.

GLMath. Prior to iOS 5, nearly every project needed their own math library with basic projection vector and matrix. Now with GLMath, most of the math functionalities are prepared for the developers.

GLKTextureLoader. This class is the most useful one for our goal to combine OpenGL and ClassX. GLKTextureLoader makes it much easier to load images as textures to be used in OpenGL. The developers dont have to write a complicated codes dealing with different image formats. A single function call will load a texture successfully.

A.1 GLKView

To get started with OpenGL ES 2.0, the first step we need to follow is to add a subview to the window that we can draw with OpenGL. Previously, we need to write nasty codes like creating a render buffer and frame buffer, etc to get this working. But now its simple with a new GLKit class called GLKView. If we want to use OpenGL rendering inside a view, you simply add a GLKView which is essentially a subclass of UIView and configure a few properties on it. You can then set a class as the GLKViews delegate. Once it needs to be drawn, and it will call a method on that class. In this method we can fill in our OpenGL commands.

Before using GLKit, we need to add a few frameworks to our project.

Step 1: Select the project in the Project Navigator.

Step 2: select the ClassXMobile target.

Step 3: select Build Phases, Expand the Link Binary With Libraries section, and click the Plus button.

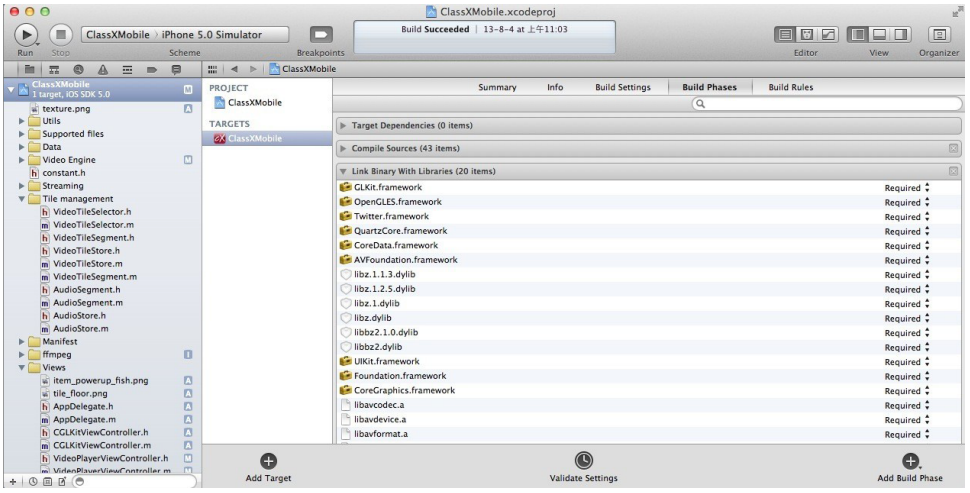


FIGURE A.1: Add required framework to the project

Step 4: From the list, select the following frameworks and click Add: QuartzCore.framework, OpenGLES.framework and GLKit.framework.

In file AppDelegate.h, add the declaration of the header file for GLKit as follows:

```
#import <GLKit/GLKit.h>
```

A.2 Creating Cubic by OpenGL

In order to make this project compatible for the future use - cubic projection implementation, I start with a nice and simple task by rendering a cubic to the screen. The basic unit of cubic is square. Since when rendering geometry with OpenGL, we cant render squares. However, OpenGL can render triangles. As a result, we can create a square with two triangles. Position defined the location of each vertex of the triangles. Indices catched the vertex of each square and make a complete cubic. The codes are as follows:

```
typedef struct {
```

```
    float Position[3];
    float Color[4];
    float TexCoord[2];
    float Normal[3];
} Vertex;

const Vertex Vertices[] = {
    // Front
    {{1, -1, 1}, {1, 0, 0, 1}, {1, 0}, {0, 0, 1}},
    {{1, 1, 1}, {0, 1, 0, 1}, {1, 1}, {0, 0, 1}},
    {{-1, 1, 1}, {0, 0, 1, 1}, {0, 1}, {0, 0, 1}},
    {{-1, -1, 1}, {0, 0, 0, 1}, {0, 0}, {0, 0, 1}},
    // Back
    {{1, 1, -1}, {1, 0, 0, 1}, {0, 1}, {0, 0, -1}},
    {{-1, -1, -1}, {0, 1, 0, 1}, {1, 0}, {0, 0, -1}},
    {{1, -1, -1}, {0, 0, 1, 1}, {0, 0}, {0, 0, -1}},
    {{-1, 1, -1}, {0, 0, 0, 1}, {1, 1}, {0, 0, -1}},
    // Left
    {{-1, -1, 1}, {1, 0, 0, 1}, {1, 0}, {-1, 0, 0}},
    {{-1, 1, 1}, {0, 1, 0, 1}, {1, 1}, {-1, 0, 0}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}, {-1, 0, 0}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}, {-1, 0, 0}},
    // Right
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}, {1, 0, 0}},
    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}, {1, 0, 0}},
```

```
    {{1, 1, 1}, {0, 0, 1, 1}, {0, 1}, {1, 0, 0}},
    {{1, -1, 1}, {0, 0, 0, 1}, {0, 0}, {1, 0, 0}},
    // Top
    {{1, 1, 1}, {1, 0, 0, 1}, {1, 0}, {0, 1, 0}},
    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}, {0, 1, 0}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}, {0, 1, 0}},
    {{-1, 1, 1}, {0, 0, 0, 1}, {0, 0}, {0, 1, 0}},
    // Bottom
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}, {0, -1, 0}},
    {{1, -1, 1}, {0, 1, 0, 1}, {1, 1}, {0, -1, 0}},
    {{-1, -1, 1}, {0, 0, 1, 1}, {0, 1}, {0, -1, 0}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}, {0, -1, 0}}
};

const GLubyte Indices[] = {
    // Front
    0, 1, 2,
    2, 3, 0,
    // Back
    4, 6, 5,
    4, 5, 7,
    // Left
    8, 9, 10,
    10, 11, 8,
    // Right
```

```
    12, 13, 14,
    14, 15, 12,
    // Top
    16, 17, 18,
    18, 19, 16,
    // Bottom
    20, 21, 22,
    22, 23, 20
};
```

A.3 Creating Vertex Buffer Object

The next step is to send the data to OpenGL. By the OpenGL Vertex buffer objects, we call few functions to send the data to OpenGL.

So in method setupGL, the critical thing it does is set the current OpenGL context to the current context. This is important in case some other code has changed the global context.

Once we prepare to draw, we have to tell OpenGL which vertex buffer objects we will use. So we need to bind the vertex and index buffers.

Next, we use the glEnableVertexAttribArray to enable three attributes, the vertex position, the vertex color, and the texture position. GLKit has predefined constants we need to use for these GLKVertexAttribTexCoord0, GLKVertexAttribTexCoord1, GLKVertexAttribPosition and GLKVertexAttribColor. Next,

we call glVertexAttribPointer to feed the correct values to these variables for the vertex shader.

The usage of parameters is explained as follows:

The first parameter specifies the attribute name.

The second parameter specifies how many values are present for each vertex. For example, in Vertex struct, we can see that for the position there are three floats (x,y,z), so the number here is 3.

The third parameter specifies the type of each value.

The fourth parameter is always set to false.

The fifth parameter is the size of the data structure containing the per-vertex data. So we can use the function sizeof here to get the compiler to compute it for us.

The final parameter is the offset within the structure to find this data.

```
- (void)setupGL {
    [EAGLContext setCurrentContext:self.context];
    glEnable(GL_CULL_FACE);
    self.effect = [[GLKBaseEffect alloc] init];
    NSDictionary * options = [NSDictionary
  dictionaryWithObjectsAndKeys:
                                [NSNumber numberWithBool
  :YES],

  GLKTextureLoaderOriginBottomLeft,
```

```
                                nil];
  NSError * error;
  NSString *path = [[NSBundle mainBundle]
pathForResource:@"tile_floor" ofType:@"png"];
  GLKTextureInfo * info = [GLKTextureLoader
textureWithContentsOfFile:path options:options
error:&error];
  if (info == nil) {
      NSLog(@"Error loading file: %@", [error
localizedDescription]);
  }
  self.effect.texture2d0.name = info.name;
  self.effect.texture2d0.enabled = true;
  path = [[NSBundle mainBundle] pathForResource:@"
texture" ofType:@"png"];
  info = [GLKTextureLoader textureWithContentsOfFile
:path options:options error:&error];
  if (info == nil) {
      NSLog(@"Error loading file: %@", [error
localizedDescription]);
  }
  self.effect.texture2d1.name = info.name;
  self.effect.texture2d1.enabled = true;
  self.effect.texture2d1.envMode =
GLKTextureEnvModeDecal;
```

```
 glGenVertexArraysOES(1, &_vertexArray);
 glBindVertexArrayOES(_vertexArray);
 glGenBuffers(1, &_vertexBuffer);
 glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
 glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices),
Vertices, GL_STATIC_DRAW);
 glGenBuffers(1, &_indexBuffer);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer
);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(
Indices), Indices, GL_STATIC_DRAW);
 glEnableVertexAttribArray(GLKVertexAttribPosition)
;
 glVertexAttribPointer(GLKVertexAttribPosition, 3,
GL_FLOAT, GL_FALSE, sizeof(Vertex), (const GLvoid
*) offsetof(Vertex, Position));
 glEnableVertexAttribArray(GLKVertexAttribColor);
 glVertexAttribPointer(GLKVertexAttribColor, 4,
GL_FLOAT, GL_FALSE, sizeof(Vertex), (const GLvoid
*) offsetof(Vertex, Color));
 glEnableVertexAttribArray(GLKVertexAttribTexCoord0
);
 glVertexAttribPointer(GLKVertexAttribTexCoord0, 2,
 GL_FLOAT, GL_FALSE, sizeof(Vertex), (const GLvoid
*) offsetof(Vertex, TexCoord));
```

```
    glEnableVertexAttribArray(GLKVertexAttribTexCoord1
);
    glVertexAttribPointer(GLKVertexAttribTexCoord1, 2,
    GL_FLOAT, GL_FALSE, sizeof(Vertex), (const GLvoid
*) offsetof(Vertex, TexCoord));
    glEnableVertexAttribArray(GLKVertexAttribNormal);
    glVertexAttribPointer(GLKVertexAttribNormal, 3,
GL_FLOAT, GL_FALSE, sizeof(Vertex), (const GLvoid
*) offsetof(Vertex, Normal));
    glBindVertexArrayOES(0);
    }
```

A.4 Rendering the GLtexture On the Cubic

To make objects appear 3D on a 2D screen, we need to apply a projection transform on the objects. Heres a diagram that shows how this works:

Basically we have two planes, a near plane is close to us and a far plane is far from us. The objects we want to display are between these two planes. The closer an object is to us we scale it so it looks smaller, and the closer the object is to the far plane we scale it so it looks bigger. This is similar to the way a human eye works.

GLKit provides you with some handy functions to set up a projection matrix. The one we are using allows us to specify the field of view along the y-axis, the aspect ratio, and the near and far planes:

The field of view is similar to camera lenses. A small field of view magnifies images by making them closer to us. A large field of view is like a wide angle lens it makes everything seem farther away. The aspect ratio is the aspect ratio you want to render to (i.e. the aspect ratio of the view). It uses this in combination with the field of view (which is for the y-axis) to determine the field of view along the x-axis.

By GLKMatrix4MakePerspective in the GLKit math library, we can easily create a perspective matrix for us all we have to do is pass in the parameters discussed above. We set the near plane to 2 units away from the eye, and the far plane to 10 units away. Aspect is the aspect ratio of the GLKView.

We need to set one more property now the modelViewMatrix. The modelViewMatrix is the transform that is applied to any geometry that the effect renders.

To make the square rotate, we add _rotation variable and change it by units per update.

Note that iOS 5 provide a very powerful function to deal with the reading of GLtexture, by GLKTextureLoader we can make UIImage buffer image into GLtexture buffer by only one single call.

Finally, we set the model view matrix on the effects transform property.

```
- (void)update {


    NSDictionary * options = [NSDictionary
   dictionaryWithObjectsAndKeys:
```

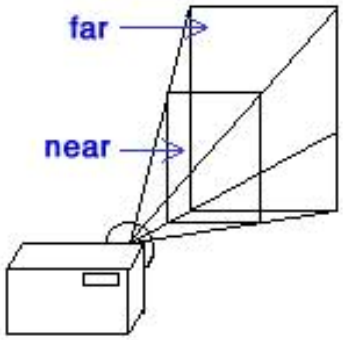


FIGURE A.2: The projection demonstration

```
                                [NSNumber numberWithBool
:YES],

GLKTextureLoaderOriginBottomLeft,
                                nil];


 NSError * error;
 CGImageRef image = _framebuffer.CGImage;
 GLKTextureInfo * info = [GLKTextureLoader
textureWithCGImage:image options:options error:&
error];
 self.effect.texture2d1.name = info.name;
 self.effect.texture2d1.enabled = true;
 self.effect.texture2d1.envMode =
GLKTextureEnvModeDecal;
 if (mode_switch){
```

```
    float aspect = fabsf(self.view.bounds.size.width /
    self.view.bounds.size.height);
    GLKMatrix4 projectionMatrix =
   GLKMatrix4MakePerspective(GLKMathDegreesToRadians
   (30.0f), aspect, 2.0f, 10.0f);
    self.effect.transform.projectionMatrix =
   projectionMatrix;



    GLKMatrix4 modelViewMatrix =
   GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);
    _rotation += 15 * self.timeSinceLastUpdate;
    modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix
   , GLKMathDegreesToRadians(25), 1, 0, 0);
    modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix
   , GLKMathDegreesToRadians(_rotation), 0, 1, 0);
        self.effect.transform.modelviewMatrix =
   modelViewMatrix;
    }
}
```