NTNU
Norwegian University of
Science and Technology

Improving the Visual Experience When Coding Computer Generated Content Using the H.264 Standard

Nicolai Berthelsen

Master of Science in Communication Technology
Submission date: June 2011
Supervisor: Tor Audun Ramstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Name of student:
Nicolai Berthelsen

Problem description:
The current methods for video compression (e.g. H.264) has been developed for natural video and thus has many weaknesses when it comes to compression of computer graphics. Typically, these methods are unsuitable for the reproduction of sharp edges. In many cases, one also experience that large uniform areas are exposed to coding artifacts, especially after a large image update. The alternative to using methods such as H.264 for the compression of PC graphics are other types of algorithms such as vector-representation. In applications that compress both natural video and computer graphics (e.g. video conferencing), it is however desirable to use the same compression method to all video material. A possible procedure is to modify an algorithm such as H.264 to better compress computer graphics. Such methods may also be considered for the H.265 standardization, as it is expected that H.265 will have many similarities with H.264. These changes , however, should not be so large that it appears as two different algorithms within the same standard.

The task is to examine if moderate changes to the H.264 standard for video compression may be done to compress computer graphics in a better way. For simulations, an H.264 reference software called JM17.2 may be used. The first part of the task is to identify visual artifacts caused when compressing computer graphics. Then, various methods that may reduce these artifacts will be investigated. One such method may be to code blocks in the pixel-domain instead of the transform domain.

Assignment given: 15.01.2011

Supervisor: Professor Tor Audun Ramstad

Abstract

The purpose of this Master thesis was to improve the visual experience when coding *computer generated content* (CGC) using the H.264 standard. As H.264 is designed primarily to code natural video, it exhibits weaknesses when coding CGC at low bit rates. The thesis has focused on identifying and modifying the components in the H.264 algorithm responsible for the occurrence of unwanted noise artifacts.

The research method was based on performing quantitative research to confirm or deny the hypothesis claiming that the H.264 algorithm performs sub-optimally when coding CGC. Experiments were conducted using coders written specifically for the thesis. The results from these experiments were then analyzed, and conclusions were drawn based on empirical observations.

An implementation of H.264 was used to identify the noise artifacts resulting from coding CGC at low rates. The results indicated that H.264 indeed performs sub-optimally when coding CGC. We learned that the reason for this was that the characteristics of CGC led to the signal being more compactly represented in the spatial domain than in the transform domain. We therefore proposed to omit the component transform and quantize the residual signal directly. This method, called *residual scalar quantization* (RSQ), was shown to outperform traditional H.264 coding for certain CGC in terms of quantified visual quality and bit rate. However, even when outperformed, the RSQ coder did not exhibit any of the noise artifacts present when coding with the traditional coder. We also introduced Rate-Distortion optimization, which allowed the coder to adaptively choose between traditional and RSQ coding, ensuring that each block is coded optimally, independent of the source content. This scheme was shown to outperform both stand-alone coders for all sample content. A quantizer with representation levels tailored specifically for the characteristics of CGC was also presented, and experiments showed that it outperformed uniform quantization when coding CGC.

The results in this thesis were produced by simplified versions of the actual coders, and may not be completely accurate. However, the accumulated results indicate that RSQ may indeed outperform traditional H.264 coding for CGC. To confirm the theories that have been presented, the proposed techniques should be implemented in a full-scale implementation of H.264 and the experiments repeated.

Keywords: H.264, residual scalar quantization, computer generated content, natural content, Rate-Distortion optimization

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) as a part of my Master of Science in Communication Technology, Signal Processing. The thesis is a cooperation between NTNU and Cisco Systems Norway. My supervisors throughout this project has been Professor Tor Audun Ramstad from NTNU and Arild Fuldseth from Cisco Systems Norway.

I would like to thank both Tor and Arild for providing good guidance, in addition to Øyvind Hansen at Cisco Systems Norway for introducing me to the possible thesis topics and working on my behalf to allow me to write this thesis for Cisco. I would also like to thank Giuseppe Ridinò for answering questions about his Huffman code implementation for MATLAB, my fellow students Magnus Ask, Zdenko Tudor and Christer-Andr Larsen for helping me by discussing ideas and problems, and the rest of my fellow students who I shared an office with and have helped make this semester unforgettable. A special thanks to Ingrid Hjulstad for providing invaluable advice and for always being available to give advice and moral support.

Trondheim June 7, 2011,
Nicolai Berthelsen

Contents

List of Figures

List of Tables

Acronyms

CGC - Computer generated content

AVC - Advanced Video Coding

RAW - Unprocessed video/images

VCL - Video Coding Layer

NAL - Network Abstraction Layer

HD - High Definition resolution

RGB - Red, Green, Blue additive color model

CABAC - Context-Adaptive Binary Arithmetic Coding

CAVLC - Context-Adaptive Variable Length Coding

ROI - Region of Interest

SAD - Sum of Absolute Difference

PSNR - Peak Signal to Noise Ratio

RDO - Rate-Distortion optimization

CIF - Common Intermediate Format

AVI - Audio Video Interleave

DCT - Discrete Cosine Transform

GOP - Group of Pictures

QP - Quantization Parameter

pdf - Probability-density function.

dB - Decibel

DPCM - Differential Pulse Code Modulation

RSQ - Residual Scalar Quantization

RBSP - Raw Byte Sequence Payload

SEI - Supplemental Enhancement Information

PDF - Portable Document Format

SSIM - Structured Similarity

MSE - Mean Square Error

Definitions

Here we list the definitions we have used in the thesis. Definitions marked with a '*' denotes definitions that have been made by the author.

- **H.264** - Video coding standard
- **YCrCb** - Family of color spaces. A way of encoding *RGB* information
- **YUV** - Raw video format. Used interchangeably with *YCrCb*
- **DC** - Mean value of waveform
- **Isotropy** - Isotropy means uniformity in all directions
- **Anisotropy** - Anisotropy is the property of being directionally dependent
- **Artificial content*** - Content generated on a computer, e.g. text and graphical objects
- **Compound content*** - Artificial content with embedded natural images/video
- **4:2:x** - Color downsampling scheme
- **CIF** - 352x288 resolution
- **Initial RSQ coder*** - The RSQ coder using uniform quantization
- **End-zone RSQ coder*** - The RSQ coder using extended end-zone quantization
- **Dyadic** - A dyadic number is defined as 2 to the power of a natural number b (i.e. 2b).

Chapter 1

Introduction

H.264/Advanced Video Coding (AVC), also referred to as *MPEG-4 Part 10* is an open video coding standard, created as a joint effort by ITU-T *Video Coding Experts Group* and the ISO/IEC *Moving Picture Experts Group*. It is a successor to earlier standards such as MPEG-2 (commonly used in DVD and Digital Video broadcasting) and H.263 (commonly used in videoconferencing). Its use encompasses areas such as the compression of HD resolution (most commonly 1280x720 or 1920x1080 pixel) video for Blu Ray[4], video streaming such as Youtube (`www.youtube.com`) or Vimeo (`www.vimeo.com`), file storage on computers, and videoconferencing. It is currently one of the most commonly used formats for video compression.

1.1 Scope

This thesis will focus on the use of the H.264 algorithm to code *computer generated content* (CGC) at low bit rates, and the noise and artifacts that is the result of this low rate coding. In this context, CGC refers to video and images generated on a computer. "Coding content" refers to the process of reducing the information or data that has to be represented or transmitted in terms of bits, also known as data compression. Compressing information usually involves the loss of signal accuracy, which in turn leads to reduced visual quality. It is therefore important to compress information such that the removed data is perceptually insignificant.

Examples of CGC includes *PowerPoint* presentations, remote desktop viewing and demo recordings. As these areas of use become increasingly important in our daily lives, the importance of being able to compress the CGC whilst sustaining a satisfactory level of fidelity also increases. However, there are known difficulties when compressing computer graphics using H.264

or similar coding standards. These difficulties will be addressed in Section 1.2.

We will classify content that consists of purely computer graphics (i.e. text and graphical objects) as *artificial content*. The other classifications we will use is *compound content* which refers to artificial content with embedded natural content. CGC will be used interchangeably with artificial content.

1.2 Motivation

H.264 is designed primarily to compress natural video, i.e. content generated by video cameras [17]. This presents a problem when the source to be encoded consists of computer graphics, because the characteristics of computer graphics are inherently distinct from natural video, which we will investigate further in Section 2.3. Therefore, the result of the coding process when coding CGC will be noise and artifacts that normally are not visible, even at equal or higher bit rates compared to the coding of natural video (see Section 3.4). We thus need to find ways to improve the coding process, such that the quality of coded screen content does not suffer from noise, without increasing the amount of bits (rate) needed to represent the content. Another issue we will investigate is artificial content with embedded natural images, denoted *compound content* as mentioned in the previous section, which forces us to investigate a coding scheme that allows for optimal coding of both artificial content and natural content in a combined manner.

1.3 Objectives

This section will identify the objectives of the thesis. As we have discussed, the main objective will be to improve upon the H.264 coding algorithm such that we achieve better results when coding content that is generated on a computer. This main objective will be split into several smaller objectives to make them more manageable. These objectives are:

1. Analyze the characteristics of natural video and CGC, and learn how they differ from one another.

2. Learn how the H.264 coder actually performs when coding CGC at low bit rates.

3. Classify the noise and artifacts that occurs when coding CGC using the regular H.264 algorithm.

4. Identify which components in the H.264 algorithm that causes the sub-optimal performance when coding CGC.

5. Investigate how these components might be modified and improved to better take into consideration the characteristics of the CGC.

6. Introduce a method that determines whether the source that is to be coded consists of natural video, artificial content or a mix of the two (compound content). Then code the content accordingly.

1.4 Methodology

As mentioned earlier, the purpose of the thesis is to modify the current H.264 algorithm such that we may improve upon the results when coding CGC. To accomplish this, we will first need to use existing tools to encode CGC while using a low bit rate, to ensure the occurrence of noise and artifacts. The next step will be to identify the components that cause these artifacts. These components must then be modified and tested to learn whether or not we are able to perform coding of CGC without the occurrence of the previously identified artifacts.

Different coders were written with the purpose of testing both existing and proposed techniques and comparing their performance. The purpose of this is to make conclusions regarding the proposed methods based on empirical observations. All coders will be written in MATLAB, as MATLAB is well suited for image and video processing. To be able to compare the results we achieve when coding, we will use PSNR (further explained in Section 2.4) as a visual quality metric and the amount of bits used to represent each coded frame (i.e. the bit rate). Our desire is to achieve the best possible visual quality for the lowest possible bit rate. Hence, if the attempted improvements are successful, we should achieve an increased PSNR value for an equal or lower bit rate when using the proposed coder to code artificial content compared to using the traditional H.264 algorithm.

1.5 Outline

This section will describe the outline of the thesis.

- **Chapter 2** - We will start by reviewing the H.264 coding standard, with particular focus on the *Video Coding Layer*. Further in this chapter, we will look at the difference between natural video and artificial

content, in addition to a metric to validate visual quality and finally we will introduce *Rate-Distortion* optimization.

- **Chapter 3** - This chapter contains a brief description of the tools and parameters used to record and code the videos used for all experiments in this thesis. Additionally, it contains a description of the content of the example videos. Finally, it will discuss the results from coding an example artificial content video sequence using an implementation of the H.264 standard called the *JM reference software*, and compare these results to the results obtained when coding a natural video sequence.

- **Chapter 4** - In this chapter we introduce a technique designed to improve upon the coding of computer graphics. It will also account for the implementation of this method, and the results from the experiments conducted using this technique.

- **Chapter 5** - Here, we will describe the implementation of an adaptive coder which is designed to optimally select the best coded block. The results from the experiments conducted using this coder are also presented and discussed in this chapter.

- **Chapter 6** - In this chapter we attempt to improve upon the quantization method used in previous chapters. It contains a detailed account of how the quantization is done, in addition to how and why it was modified. It also contains a comparison between regular uniform quantization and the proposed quantization method. Lastly it discusses the results accumulated when implementing the proposed quantization method into the adaptive coder.

- **Chapter 7** - This chapter contains the conclusion.

- **Chapter 8** - This chapter presents suggested future work.

- **Appendix A** - Here we present an overview and a short description of all the tools used in the thesis.

- **Appendix B** - This appendix contains an overview of the parameters used when coding videos with the JM reference software.

- **Appendix C** - Here we list and give a short description of all the attached videos, images and coders we have used in this thesis.

- **Appendix D** - Finally, we add the MATLAB code for one of the coders that was written for this thesis. The rest of the MATLAB coders used may be found attached with the thesis.

  The structure of the thesis is such that we present the proposed techniques, we then conduct experiments using the implemented techniques, and analyze the results. Finally we present any possible further improvements. For instance, in Chapter 3, we look at the results when coding CGC using the JM reference software. These results are then analyzed, and the suggested improvements leads us to introducing residual scalar quantization (RSQ) in Chapter 4. Again, we conduct experiments using the RSQ coder, analyze the results and suggest improvements based on these. The discovery of components causing problems when coding leads us to new improvements, like the adaptive coder suggested in Chapter 5 and the extended end-zone quantizer presented in Chapter 6.

Chapter 2

Background

This chapter will introduce and explain the mechanics of the H.264 coding standard that are relevant for the thesis. For this reason, lossless coding methods such as entropy coding will not be covered in-depth. The majority of information presented in this section is taken from [42]. The primary focus will be on the transform and quantization aspects of the algorithm. The reason why we choose to provide more in-depth information about these topics, is that they are essential to the occurrence of noisy artifacts when coding artificial content, which is the central theme of the thesis. H.264 also contains several different profiles which are designed to target specific classes of applications. These will not be investigated further in the thesis. If the reader is interested in a detailed description about profiles in H.264, this may be found in [42].

2.1 Network Abstraction Layer

As H.264 is designed to be extensively flexible and customizable, it has incorporated a mechanism that allows the source content to efficiently be coded, and then formatted into a bit stream ready for transmission. These mechanisms, or layers, are known as the *Network Abstraction Layer* (NAL) and the *Video Coding Layer* (VCL). The NAL will be briefly explained in this section, but our main focus will be on the VCL. This is because the VCL is the layer that performs the actual data compression, and hence is the layer that introduces loss of information. This layer will be where we have to introduce modifications to improve upon already existing techniques.

The NAL divides the coded data into multiple NAL units. Figure 2.1 shows how the NAL unit is structured. The first byte (NAL Unit Header) is a header byte that indicates the type of data which is contained in the NAL

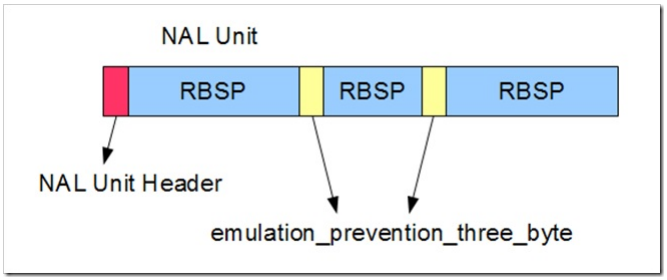

Figure 2.1: NAL unit, taken from [13]

unit. The fields named RBSP (Raw Byte Sequence Payload) contain the payload data indicated by the header. The *emulation prevention bytes* are inserted as needed into to the payload data to prevent *start code prefix* data patterns from generating, which are the patterns that prefixes each NAL unit. Each NAL unit is classified as VCL and non-VCL NAL units. The VCL NAL units contains the actual coded video samples, and the non-VCL NAL units contains any additional information. This additional information may for instance be information such as *parameter sets*, which is information that is rarely expected to change. These parameter sets contain information regarding the decoding of VCL NAL units, either sequentially coded frames or individual frames.

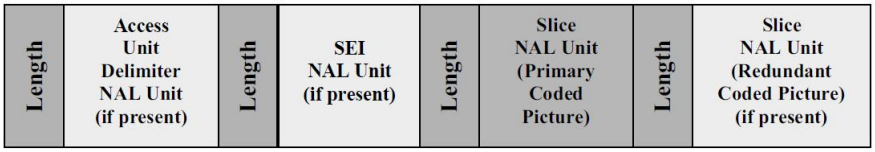

Figure 2.2: NAL access unit, taken from [41]

H.264 combines the NAL units into specified structures which are denoted *access units*, as depicted in Figure 2.2. The *Length* fields contains bytes (usually four) that denotes the size of the frame. The *Access Unit Delimiter* contains a byte sequence that is used for locating the start of the access unit. The *Supplemental Enhancement Information* (SEI) contains data such as picture timing information. The *Primary Coded Picture* is composed of a set of VCL NAL units and thus contains the coded samples of the video. The *Redundant Coded Picture* contains identical representations of areas of the same video frame which was coded in the Primary Coded Picture

block. This redundant information is attached to protect from errors and loss of information when transmitting the data stream. All blocks except the Primary Coded Picture block are optional and does not necessarily need to be included in the bit stream.

2.2 Video Coding Layer

This section will explain the major features of the VCL, which is designed to efficiently represent video content. Each picture in a video sequence is coded by using both inter-frame prediction (prediction between subsequent frames) to exploit temporal redundancy, as well as intra-frame prediction (prediction between neighboring pixels/blocks within a single frame) and transform coding of the prediction residual to exploit spatial redundancy. Redundant information in images is a result of limitations in the human visual system, as well as information redundancy between consecutive frames and between neighboring pixels (temporal and spatial redundancy respectively). The color components in an image is also subject to the limitations of human perception and therefore also prone to be subject for data compression, which will be explained in further detail later.

2.2.1 Chroma downsampling

The color components in an image are modeled using a color space. The most common model is the *RGB* space, where the desired color is represented as a sum of different intensities of red (R), green (G) or blue (B) color. Another color space, most commonly used in video compression, is the *YCbCr* space, also denoted *YUV* space, where each component is a sum of scaled R, G, and B components. Equation 2.1 shows the process of the conversion and are taken from [15].

$$\begin{array}{rcrcrcrcl} Y & = & (0.257 \times R) & + & (0.504 \times G) & + & (0.098 \times B) & + & 16 \\ Cr & = & (0.439 \times R) & - & (0.368 \times G) & + & (0.071 \times B) & + & 128 \\ Cb & = & -(0.148 \times R) & - & (0.291 \times G) & + & (0.439 \times B) & + & 128 \end{array} \quad (2.1)$$

The inverse operation is given by:

$$\begin{array}{rcrcrcl} B & = & 1.164 \times (Y - 16) & + & 2.018 \times (Cb - 128) & & \\ G & = & 1.164 \times (Y - 16) & - & 0.813 \times (Cr - 128) & - & 0.391 \times (Cb - 128) \\ R & = & 1.164 \times (Y - 16) & + & 1.596 \times (Cr - 128) & & \end{array}$$

Component Y is called the *luma* component, and represents the bright-

ness. Cr and Cb are called *chroma* components. Cr and Cb represents the red-difference and the blue-difference respectively. As the human visual system is more sensitive to the luma component than the chroma component[42], the chroma components can be downsampled (e.g. we only transmit every other chroma component) to reduce the bit rate without significantly harming the perceived visual quality. The most common subsampling format is the 4:2:0 format, which is standard in H.264. With 4:2:0, both chroma components are downsampled by a factor of two in both horizontal and vertical directions, resulting in a total reduction in chroma components by a factor of four compared to luma components, which are not downsampled in this scheme. Figure 2.3 shows one variant of 4:2:0 chroma downsampling. We

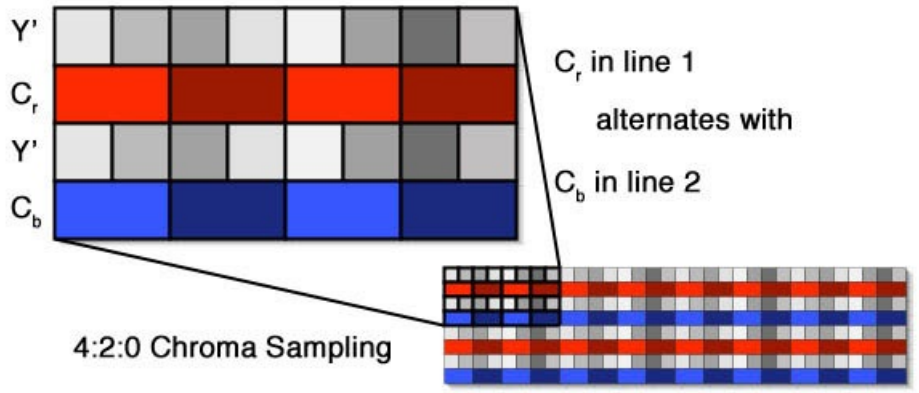

Figure 2.3: 4:2:0 chroma downsampling, taken from [28]

will use the abbreviation YUV interchangeably with YCrCb throughout the thesis. The convention for U and V will then be $U = Cb$ and $V = Cr$. More details on chroma downsampling can be found in [29] and [28].

2.2.2 Slices

H.264 uses a *slice-structure*, which means it divides frames into potentially smaller fragments called slices (we say potentially as a slice may fully cover a frame if desired), which are encoded and decoded independently of other slices, i.e. they are self-contained. Frames may be divided into slices of any size and shape (as long as it is within the boundaries of the frame), and may among other things be used to support so-called *region of interest* (ROI) coding, where a specific area of an image is more significant than other areas and it is therefore desirable to code this area using a higher bit rate than

when coding the rest of the image. These slices are coded mainly using three different coding types, namely *I*, *P* and *B*-slices. I-slices are solely *intra-predicted*, while P and B-slices also make use of *inter-prediction* in addition to intra-prediction. P-slices may contain forward predicted (i.e. they are predicted from preceding slices) macroblocks and B-slices may contain forward and backwards (i.e. they may be predicted from slices ahead in time) predicted macroblocks. Moreover, a new addition to H.264 compared to previous coding standards are the SP and SI slices, however, these will not be discussed in the thesis.

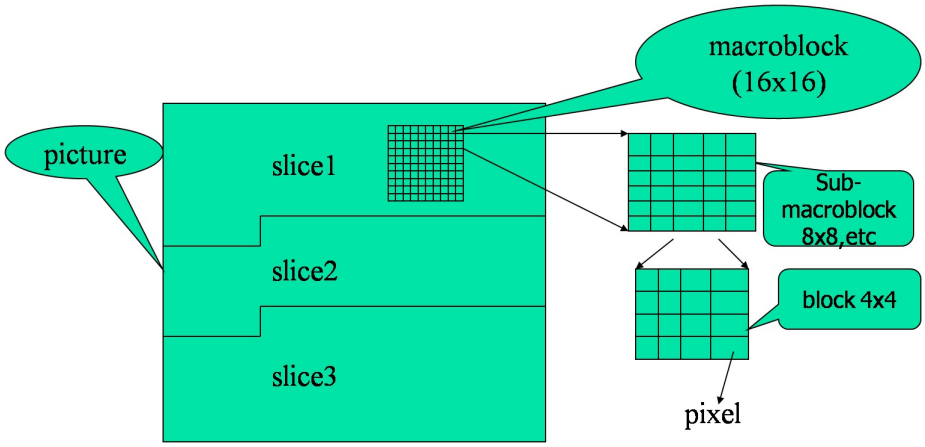

Figure 2.4: Overview of the H.264 data hierarchy, taken from [38]

Figure 2.4 depicts how H.264 splits each frame into slices. Each slice is then composed of a set of macroblocks of size 16x16 pixels. The macroblocks are composed of a set of four sub-macroblocks of size 8x8 which again is composed of a set of four blocks of size 4x4. These block compositions are denoted as levels. The level on which the coder desires to operate may be chosen independently for each macroblock.

2.2.3 Intra-prediction

Intra-prediction is the process of predicting from blocks that are contained *within* the same frame. H.264 uses intra-prediction on each macroblock, which allows us to transmit even less information, i.e. we only have to transmit the residual signal of the predicted block in addition to the prediction mode used to be able to completely reconstruct each block. The size of the intra-prediction mode used depends on the characteristics of the area that is to be coded. Small prediction modes with a size of 4x4 are used for areas that

are highly detailed. Larger prediction modes, like 16x16, are used for areas that contains less detail and are generally smoother. The intra-prediction is done such that each block is predicted using previously encoded and reconstructed samples from above and to the left, as depicted in Figure 2.5, illustrated as shaded blocks. These samples must be previously encoded to be available for use in the different prediction modes. If they are not previously coded or they belong to another slice, they cannot be used for prediction (as slices must be self-contained). The samples used for prediction, denoted by P are formed from the weighted average of the adjacent pixels which are indicated by the arrows. The Sum of Absolute Difference (SAD)[1] is then calculated for each mode, and the mode which achieves the lowest SAD value is considered to be the best match.

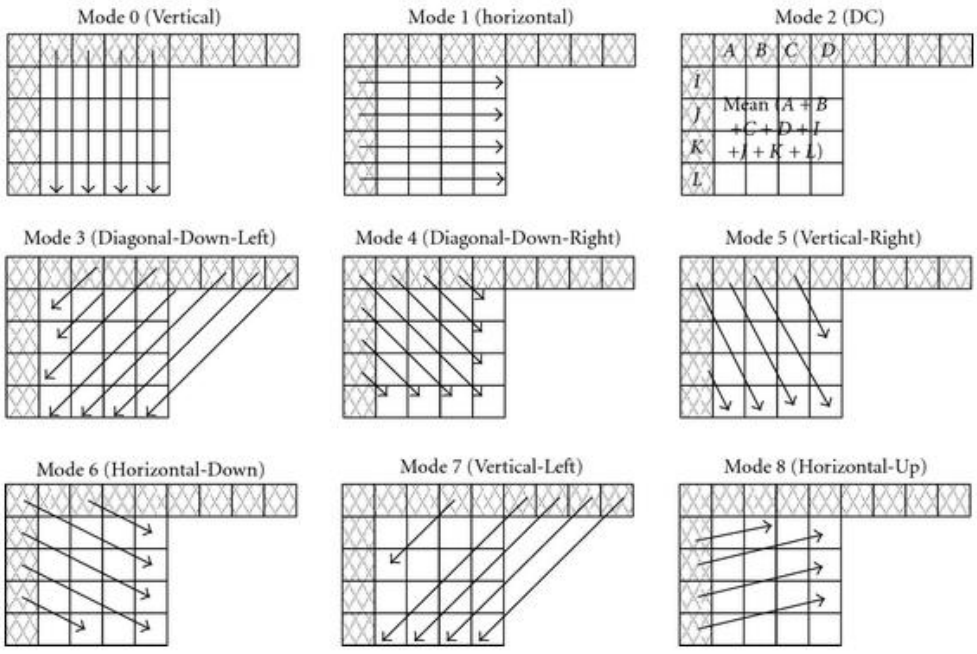

Figure 2.5: H.264 4x4 intra-prediction modes, taken from [11].

After finding the best match, the predicted block P is set to the best matching block. P is then subtracted from the current block, resulting in a residual signal. This residual signal is then transmitted along with the

[1]SAD is a metric for measuring the similarity between blocks, by taking the absolute difference between each pixel and summing these differences. It is commonly used because of its simplicity, which ensures that its quick.

prediction mode used. For 16x16 intra-prediction, there are four available prediction modes, which are shown in Figure 2.6.

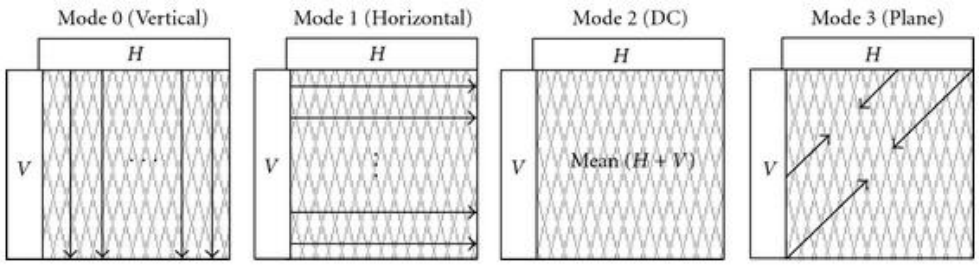

Figure 2.6: H.264 16x16 intra-prediction modes, taken from [11].

We see that the first three 16x16 prediction modes are similar to the first three 4x4 prediction modes. The fourth mode uses a linear "plane" function which is fitted to the upper and left-hand samples H and V [33]. This mode excels when used on areas with smoothly-varying luminance.

2.2.4 Inter-prediction

Inter-prediction is the process of predicting blocks *between* consecutive frames. It is used to exploit temporal redundancy between neighboring frames. H.264 uses block-based motion compensation, as previous standards, but with the addition of multiple optional block sizes and up to quarter-pixel motion vectors[32]. The optional block sizes are defined such that the coder chooses a size for the macroblock to be predicted, which may range from 16x16 to 4x4. The macroblock may then be sub-divided into even smaller blocks, or partitions, down until a block size of 4x4. The prediction is then done in a similar fashion as the intra-prediction, i.e. we compare the block to be predicted to all blocks within a predetermined search area by calculating the SAD for each block. The block with the lowest SAD is considered to be *the best match*, and will be used to predict the current block. The coder then subtracts the predicted block from the current block, leaving only the residuals to be coded and transmitted along with a *motion vector*. The motion vector determines the displacement of the predicted block. The coder has to code and transmit one motion-vector for each macroblock and one for each partition of the macroblock, if any. The macroblock sizes are determined based on the characteristics of the area to be coded, much like the intra-prediction scheme discussed in the previous subsection.

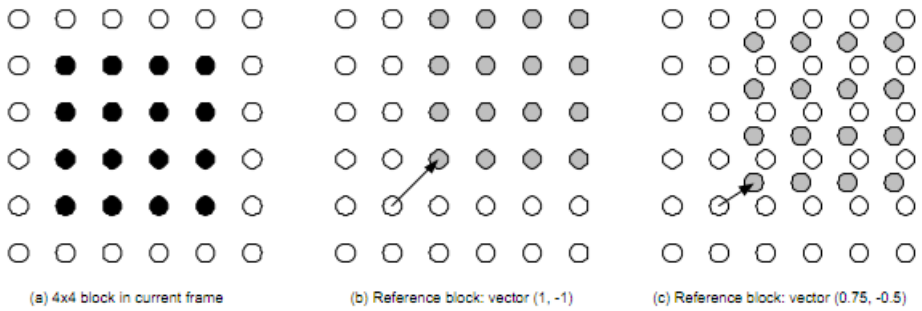

Figure 2.7: Integer and sub-pixel inter-prediction, taken from [32].

As mentioned, a new feature in H.264 compared to previous standards is the use of quarter-pixel motion compensation. Figure 2.7 shows both integer and sub-pixel prediction. Sub-pixel prediction outperforms integer-pixel prediction in terms of compression rate, at the expense of increased complexity, with quarter-pixel prediction outperforming half-pixel prediction. The sub-pixel samples are generated by interpolating between neighboring pixels, i.e. each half-pixel sample is a weighted sum of a pre-specified amount of neighboring pixels. The quarter-pixel samples are then produced by interpolating between neighboring half- or integer-pixel samples.

We will also mention a simplified inter-prediction technique called *differential coding*. Here, each frame is subtracted from an earlier anchor frame, to exploit the temporal redundancy between neighboring frames. As this technique is inherently straightforward, it is easily implemented in any testing environment, and will consequently be implemented before an implementation of a more complex motion prediction system such as the one used in H.264. A more detailed description of motion compensation can be found in [10].

2.2.5 Transform coding

Transform coding is a technique used to exploit the isotropic[2](i.e. high spatial correlation between neighboring pixels) nature of natural images, allowing us to represent the information in the image in a more compact form. This is done by decomposing the original signal to a sum of selected basis functions. The desired effect is that as much of the energy of the image as possible will be stored in a small amount of the transform coefficients, i.e. making the

[2]Isotropy means uniformity in all directions

signal representation more compact. This is possible because of Parseval's relation[30], which is given as

$$\sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{2N} \sum_{k=0}^{N-1} w(k)|X_c(k)|^2 \tag{2.2}$$

Equation 2.2 shows the energy conserving property of the 1D *Discrete Cosine Transform* (DCT)[10] function, and tells us that the energy in the spatial domain is equal to the scaled energy in the frequency domain. $x(n)$ is the signal in the spatial domain, $X_c(k)$ is the signal in the frequency domain and $w(k)$ is a weighting function given as

$$w(k) = \begin{cases} \frac{1}{2}, & k = 0 \\ 1, & k \neq 0 \end{cases}$$

Equation 2.2 is easily expanded to the case of 2D-DCT or any other transform which conserves the signal energy in the frequency domain, and it tells us that reducing the energy in the frequency domain will result in a reduction of energy in the spatial domain. This means that we can achieve compression by reducing the dynamic range of coefficients or even remove them in the frequency domain.

H.264 uses 4x4 integer transform coding, which is an approximation to the DCT used in many previous image and video coding standards. The DCT is avoided in H.264, as it uses irrational numbers, because this may result in a non-perfect reconstruction. Using a 4x4 transform instead of the previously used 8x8 transform also helps to reduce the amount of ringing artifacts, i.e. artifacts that appear as spurious signals around sharp transitions in the image. However, H.264 may still adaptively choose to use 8x8 blocks for the transform, in the presence of highly correlated regions, which results in a more efficient compression.

After the color downsampling and prediction has been done (or have been skipped), H.264 applies the component transform on the prediction residuals for all blocks of luma and chroma components. The 4x4 integer transform applied is described by the following matrix:

$$H_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}.$$

These matrix coefficients are set such that a minimum amount of bits are used

for storage, while still retaining orthogonal[3] rows. If the intra-prediction used was 16x16 as discussed in Section 2.2.3, the luma DC coefficients resulting from the primary spatial transform undergo a 4x4 Hadamard (H_2) transform to obtain better compression in smooth areas. The H_2 transform is given by:

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}.$$

Finally, a 2x2 Hadamard transform (H_3) is applied for the transform of the 4 DC coefficients of each chroma components from the primary transform. H_3 is given as:

$$H_3 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The transformed coefficients are ordered horizontally and vertically, with the frequency of the horizontal components increasing from the lowest to the highest frequency as we move from left to right in the block, and the frequency of the vertical components increasing from the lowest to highest frequency as we move downwards in the block, as depicted in Figure 2.8. Figure 2.8 shows the DCT frequency map, and not the integer transform used in H.264. But as the integer transform is an approximation to DCT, the frequency maps will be virtually similar.

Because of the DCT transform's property of concentrating frequency components to specific areas, the quantization can be done such that it removes or minimizes coefficients that are less visually significant. For natural images, these are typically the high-frequency components, as explained in [19].

2.2.6 Quantization

Quantization is the step in the encoding process that introduces signal loss. This is done by reducing the range of values that must be transmitted. However, this reduction makes perfect reconstruction impossible, as quantization is a *many-to-few mapping*, meaning that multiple values gets mapped to the same representation value. This produces a trade-off between visual quality and compression rate. An example uniform quantizer is depicted in Figure 2.9. The quantization step size (often denoted Δ) is one for this quantizer.

[3]Orthogonality refers to the property of being mutually perpendicular

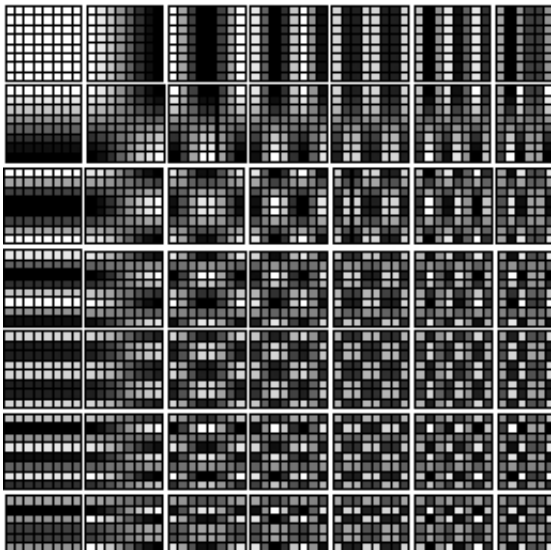

Figure 2.8: Two dimensional DCT frequencies, taken from [36].

Quantization is normally done by dividing the coefficients by a specified quantization value, as equation 2.3 shows.

$$X_q(i,j) = \text{sign}\{X(i,j)\} \left\lfloor \frac{X(i,j) + f(Q_s)}{Q_s} \right\rceil \tag{2.3}$$

Here, i and j represent the row and column indices of the frame, Q_s represents the quantization step size, and $f(Q_s)$ represents the quantization width near the origin (the dead-zone). $\lfloor \cdot \rceil$ denotes the rounding function (i.e. rounding to the closest integer), $X(i,j)$ represent an input coefficient and $X_q(i,j)$ represents the quantized coefficient.

H.264 however, avoids division to reduce computational complexity. It achieves this by using the equation given by:

$$X_q(i,j) = \text{sign}\{X(i,j)\}[(X(i,j)A(Q) + f2^L) >> L].$$

Here, the parameter Q varies from 0 to Q_{max}, where Q_{max} is derived from a quantization parameter (QP). The QP is manually set in the encoder and decides the coarseness of the quantization, i.e. a large QP results in a higher compression at the expense of reduced visual quality, whereas a low QP results in a finer quantization, yielding increased visual quality for the reconstructed image at the expense of a higher bit rate. ">>" represents a binary shift right. $A(Q)$ is set such that the finest quantization corresponds to zero,

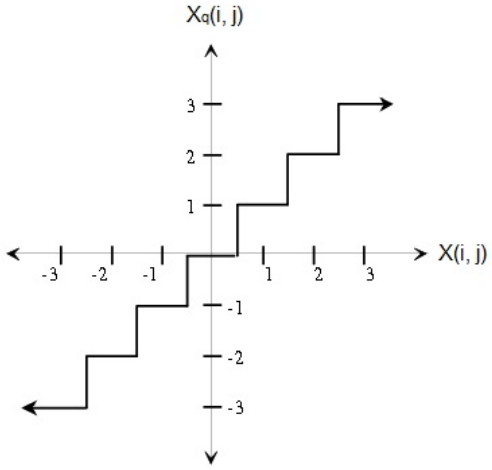

Figure 2.9: Uniform scalar quantizer with 7 representation levels. It can be viewed of as a dead-zone quantizer with a dead zone with a size of one step. Taken from [6].

and the coarsest quantization to Q_{max}. f again represents the dead-zone control parameter. 2^L represents an L-bit right shift, which is equivalent to dividing by 2^L, where L is set manually. Changes in L will vary the approximation error as well as the dynamic range of the reconstructed and quantized signal. The reconstruction process will not be thoroughly explained here. Suffice to say, the reconstruction of coefficients that are quantized using equation 2.3 is done as follows:

$$X_r(i,j) = Q_s X_q(i,j)$$

The reconstruction used in H.264 is similar, though slightly more complex. More details about the transform coding and the quantization process can be found in [22].

2.2.7 Deblocking filter

A common problem when using block-based coding is the occurrence of so-called blocking artifacts, especially for low-rate coding. Figure 2.10 shows clearly visible blocking artifacts. To counteract the occurrence of these blocking artifacts, H.264 introduces an in-loop deblocking filter. The filter investigates the difference between two samples near the edge of a block. Should this difference be larger than a threshold calculated from the quantization

Figure 2.10: Blocking artifacts resulting from low-rate coding. Taken from [43].

parameter (QP), then it is likely that we have encountered a block artifact, and the difference will be smoothed out. There is also an upper threshold to reflect the fact that the edge might be a part of the actual behavior of the image. Hence, if the difference surpasses the upper threshold, it is probably not a block artifact, but instead a natural transition between two objects or similar.

Because of the inherent characteristics of artificial content, i.e. large amounts of sharp and regular edges (which will be discussed in greater detail in Section 2.3) omitting the use of a deblocking filter when coding compound images may result in gain in visual quality, as discussed in [16].

2.2.8 Entropy coding

This section will only briefly explain the entropy coding performed in H.264, as it is irrelevant for the creation of noisy artifacts.

After the component transform and quantization, the coefficients undergo a zig-zag scan, which reorders the coefficients from a 2D matrix form to vector form. The vector is ordered such that the top-left coefficients (i.e. the low frequency components of the signal) are listed first and the bottom-right coefficients (i.e. the high frequency components of the signal) are listed last. When coding natural content, the typical zig-zag coded vector will contain large values for the low-frequency part and decreasing values for the high-frequency part. The entropy coders used in H.264, called *Context-Adaptive Variable Length Coding* (CAVLC) or *Context-Adaptive Binary Arithmetic*

Coding (CABAC) will then attempt to convey this information, using as few bits as possible based on the characteristics of the zig-zag coded block. More information about CAVLC and CABAC may be found in [42] and [23].

Figure 2.11 shows an overview of the basic H.264 coding structure. We see that the input video signal is split into macroblocks and divided into slices, which are processed via intra- and inter-prediction, transforms, quantization, a deblocking filter and finally entropy coded.

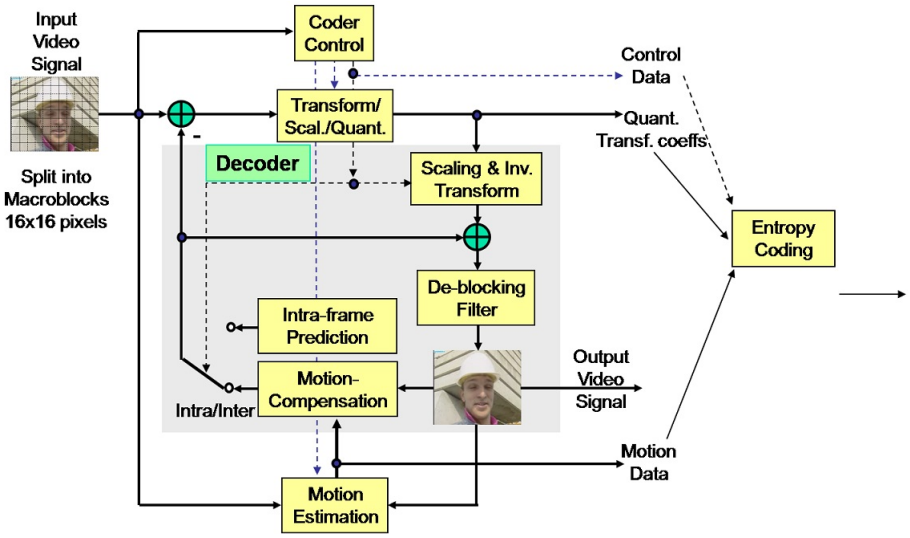

Figure 2.11: Basic H.264 coding structure. Taken from [38]

2.3 Natural video vs Computer generated content

Natural video refers to video created by recording using a video camera. Computer generated content refers to video created by recording directly from the screen of a computer, by using software tailored for this specific task. Figure 2.12 shows an example of CGC and a natural image.

Technically, there are several differences between natural video and artificial content. We expect natural video to have a large correlation between neighboring samples as we assume isotropy[16]. This correlation is then exploited using transform coding. Artificial content however, is assumed to be strongly anisotropic[4], which makes the use of transforms less efficient,

[4]Anisotropy is the property of being directionally dependent.

Figure 2.12: Computer generated content (left) and natural content (right). Taken from [26] and [3].

as is shown in Figure 2.13. We see that the resulting AC coefficients when DCT transforming artificial content are severely larger than the AC coefficients resulting from transforming natural images, which are more compactly represented.

In addition, artificial content blocks often consists of a limited amount of colors, and edges are normally sharp and regular, which translates to artificial content lending itself to be more efficiently coded in the spatial domain.

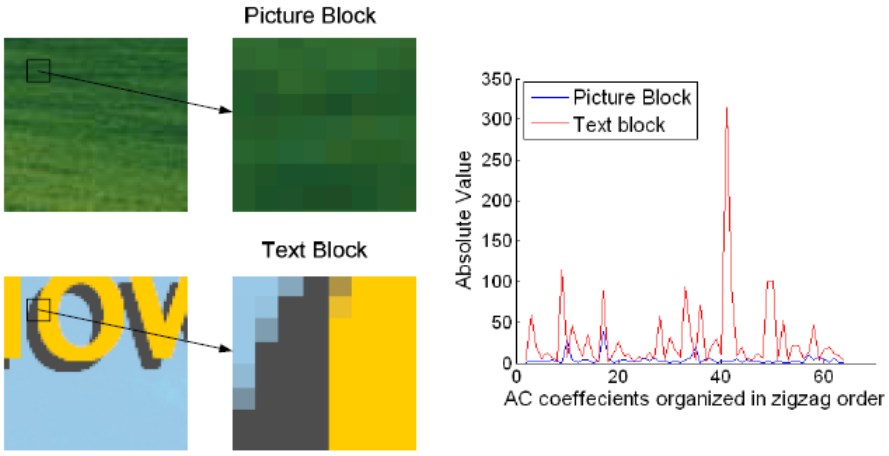

Figure 2.13: Values of AC coefficients after DCT transform, natural image and screen content, taken from [9].

Another characteristic of artificial content is that the distortion produced

may be more noticeable by a human observer. As the human eye has a relatively high tolerance of loss in natural images, lossy compression algorithms lend themselves well to the purpose of coding natural images. However, loss of information in artificial content tend to be more pronounced, such as ringing artifacts occurring from compressing high-frequency visual content[35]. Graphical objects in artificial content are often placed such that nearby objects are highly contrasted from each other, to be easily distinguishable for the user. However, when compressed, the artifacts and noise emanating from these contrasted objects will then be significantly more distinguishable compared to their natural video counterparts, which are usually less protruding.

2.4 Measuring quality using PSNR

The most commonly used measure for objective quality in reconstructed images is the *peak signal-to-noise ratio* (PSNR)[14]. PSNR is shown to perform close to subjective quality measures when comparing similar video content[12]. The PSNR is defined in Equation 2.4.

$$PSNR = 20 \log_{10} \frac{MAX^2}{MSE}. \tag{2.4}$$

Here MAX is the maximum pixel value in the image, i.e. $MAX = 2^B - 1$, where B is the amount of bits per pixel. MSE is the *Mean Square Error*, defined as:

$$MSE = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2.$$

Here I and K represents the original and reconstructed image respectively. M and N represents the height and width of the image. PSNR is measured in *decibels* (dB). We will use the PSNR quality metric in this thesis to quantify the resulting visual quality we achieve when encoding. By studying the PSNR and the amount of bits used, we can get a good idea of how well our coder performs. We may then use this information to compare our coder to other existing coders.

However, PSNR does have its weaknesses. One problem is that PSNR does not correlate well with subjective quality measures when measuring across different types of video content[12]. This makes it difficult to compare artificial content to natural videos, as the characteristics of the two distinct video types (natural and artificial content) exhibit significant differences, as explained in Section 2.3.

2.5 Rate-Distortion Optimization

This section will explain Rate-Distortion optimization (RDO), and how it may be used to optimize the coding process based on the measured visual quality and bit rate. If we assume we have two coders, one to code natural content and one to code artificial content, and we assume the source to be coded contains both natural content and artificial content, i.e. compound content. If we use only one of the tailored coders to code the compound content, this will lead to sub-optimal performance in terms of visual quality. Instead, we may ensure that the optimal coder is used by coding each block in a frame with both coders, and then compare the distortion and the bit rate per block for both coders. The best coded block is then selected for use in the reconstructed frame. This comparison may be done by using the Lagrange multiplier, shown in equation 2.5.

$$J = D + \lambda R \tag{2.5}$$

Here, D is the distortion (square error), R is the rate, and λ is a variable set to satisfy a side-condition. In our case, our choice of λ will represent how much we weigh the bit rate compared to the distortion. The coder that minimizes J will be the optimal choice for the given block. More details about using Lagrange multipliers in RDO can be found in [18]. The distortion is denoted as the square error between the original block or frame (I) and the reconstructed block or frame ($\tilde{I}$) as shown in Equation 2.6, where M and N denotes the height and width of the block.

$$D = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (I(i,j) - \tilde{I}(i,j))^2 \tag{2.6}$$

We will also use the Lagrange multiplier as a method of comparison between reconstructed images, for an easier way to compare the results of different coders in terms of the distortion and the bit rate. This will be further explained in Section 4.3.

Chapter 3

Tools And Methodology

This chapter will discuss the tools and methodology used to perform the experiments done in this thesis. We will explain how we have recorded our sample videos, how to play the .yuv files that contains the reconstructed video after coding, and how to convert files between .avi, .264 and .yuv format.

We will also introduce an open-source implementation of the H.264 standard called the *JM reference software* which we will use to evaluate the performance of the H.264 standard when coding different content. We will investigate and explain the parameters that must be used in order for the JM reference software to work correctly when coding both natural and artificial content. Lastly, we will analyze and discuss the results from coding our sample videos with the JM reference software.

A complete list of all tools used to perform the experiments and where they were downloaded from can be found in Appendix A.

3.1 Recording and playing video

CamStudio 2.0 was used to capture the artificial content used in the thesis. This software requires the resolution, frame capture rate and codec to be specified. For this experiment, 352x288 (CIF) resolution was chosen, a frame capture rate of 50 ms and *Microsoft Video 1* codec, which is explained in detail in [20]. This particular codec was selected for use from a list of codecs available on the computer used for recording. This particular codec was used as it was the only codec in the list that worked properly with CamStudio.

CIF-resolution was chosen as the video resolution, as encoding video using the JM reference software is highly time-consuming. Coding 50 frames while using inter prediction may result in coding durations of more than eight hours, even while using a relatively small resolution such as CIF, and larger

resolutions was therefore avoided when conducting experiments using the JM reference software. The frame capture rate is of no relevance to this experiment and was arbitrarily set.

The captured CGC was stored in an .avi file which had to be converted to .yuv file in order to encode/decode it using the JM reference software. The tool used to convert the .avi files to .yuv was *YUVTools 3.0*. After the conversion, the .yuv files can be properly coded and decoded using the JM reference software.

YUVTools were also used to play all .yuv files. To play the files, YUVTools requires the following parameters to be properly set:

- **Sampling format** - The sampling format determines the type of chroma downsampling. A more detailed explanation can be found in [29] (e.g. YUV 4:2:0).
- **Component order** - The order of the color components (e.g. Y, U, V).
- **Interlaced/progressive** - Interlaced/progressive determines if the frame is drawn sequentially line by line (progressive), or every odd line and every even line alternately (interlaced).
- **Packed/planar pixels** - Packed/planar determines if the Y, U, and V components are stored in a single array, i.e. packed together in a macropixel (packed) or stored in three separate arrays (one for each component) which are combined to form the image (planar). Detailed information about packed/planar pixel storage can be found in [25].
- **Image resolution** - The resolution of the image/video (e.g CIF or 720x480).

3.2 Experimental video content

This section introduces the videos that were used as sample videos for all the experiments conducted throughout this thesis. We use a total of three movies, all with unique characteristics.

- **foreman_cif** - The *'foreman'* sequence was downloaded from [2] and converted from .264 (file extension often used for H.264 video in RAW format) to .yuv by using the *ffmpeg* software[1]. Figure 3.1 shows a frame from the 'foreman' sequence. The 'foreman' sequence is classified as natural video and will be used to determine the experimental coder's

efficiency when coding natural video. The resolutions of the video is CIF resolution.

Figure 3.1: Still image from 'foreman_cif.avi'.

- **screen_content_1** - Artificial content recorded in CIF resolution. It is captured using CamStudio 2.0. It contains mostly text on white background in addition to the CamStudio window which is dragged around the screen to incorporate motion. Figure 3.2 shows a frame from the 'screen_content_1 sequence'. The video can be found attached in .avi format.

- **screen_content_2** - Compound content recorded in $720x480$ resolution. It is captured using CamStudio 2.0. The content consists of different variations of computer graphics. This includes a website depicting mostly text (`en.wikipedia.org`), a website containing text, embedded natural images and various other graphical objects (`www.vg.no`) and a PDF containing text in addition to graphical objects surrounding the text. Still images from the different types of content is also shown in Figure 3.3. The video can be found attached in .avi format.

Performing experiments while using these videos as source signals will give us an indication of how the coder performs on different source content such as pure natural videos, pure artificial content and compound content.

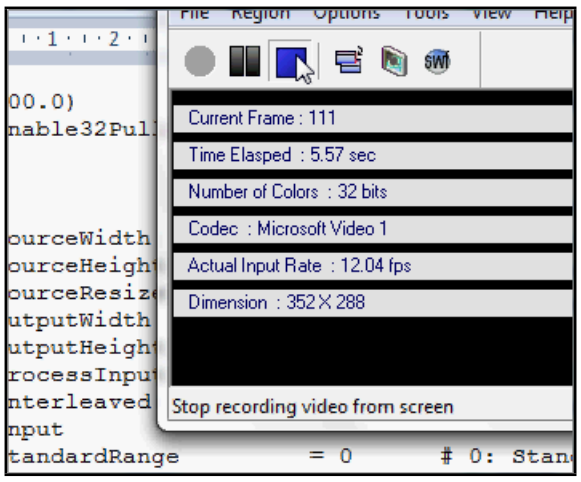

Figure 3.2: Still image from 'screen_content_1.avi'.

Figure 3.3: Still images from 'screen_content_2.avi'.

3.3 Coding with the JM reference software

The JM reference software has a large list of editable parameters, but most of these are not relevant for the thesis. For a list of the parameters that were used, see Appendix B.

We will use the quantization parameter (QP) to adjust the rate and con-

sequently the visual quality in our experiments. This is because the attempts of fixing the bit rate by setting the *RateControlEnable* parameter to 1 and setting the *Bitrate* parameter to an arbitrary value did not yield the expected results. By this, we mean that the coder would not limit itself to the given amount of bits, for unknown reasons. It therefore seemed more sensible to use the quantization parameter as a control value. This also makes our comparisons easier, as we then compare the results for each source signal with a set QP. The most important parameters will thus be the different quantization parameters (*QPISlice*, *QPPSlice*, *QPBSlice*, and *InitialQP*) which decides the degree of quantization (larger QP means coarser quantization) for the various slices (I, P, and B) and the initial frame to be coded. As mentioned in Section 2.2.6, increasing the QP will result in a reduced bit rate, but also a decrease in visual quality (PSNR).

Other parameters that must be correctly set are:

- **SourceWidth** - The width of the source video.
- **SourceHeight** - The height of the source video.
- **YUVFormat** - Determines the chroma downsampling format.
- **ProfileIDC** - Set to match the downsampling scheme selected with *YUVFormat*. In this experiment it is set to 122, to match the 4:2:2 chroma sampling that must be used in order for the YUVTools player to play the video correctly.
- **IntraPeriod** - The length of the period between I-frames (Group of Pictures(GOP)). The rest of the GOP consists of P and B frames.
- **NumberBFrames** - The amount of P and B frames within the GOP. The amount of P frames in a GOP will then be equal to $IntraPeriod - NumberBFrames$.
- **Interleaved** - Determines the use of packed or planar pixels. When coding artificial content, we must set this parameter to '1' for packed pixels, as the resulting decoded video suffers from abnormal discolored noise artifacts when using planar pixels. These artifacts are particularly pronounced in the upper-left corner of Figure 3.4.

We may also omit inter-prediction to shorten the encoding time by setting *IntraPeriod* to 1. This will increase the bit rate, but also drastically reduce the duration of the coding process.

The version of the JM reference software used in this thesis (17.2) uses a multi view coding profile (MVC), which must be disabled to avoid the

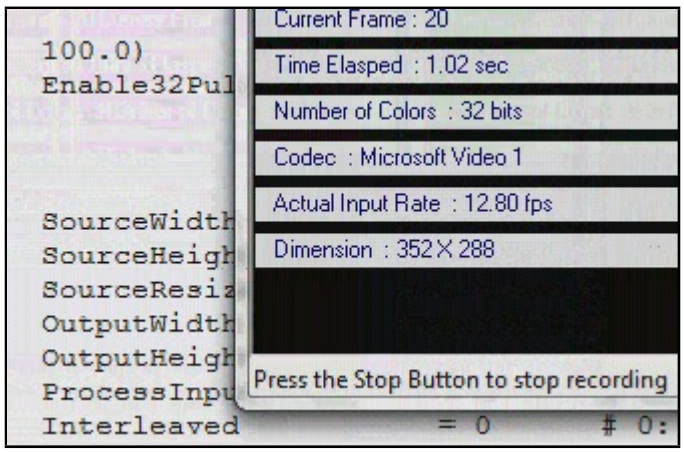

Figure 3.4: Discolored artifacts resulting from using the wrong parameters when coding with the JM reference software. Source signal is the 'screen_content_1' video.

discolored artifacts shown in Figure 3.4. This is done by inserting an empty string ("") in the fields *InputFile2* and *ReconFile2*. The parameters which were not listed in this section are set to the default value (i.e. the value used by default in the downloaded version of the JM reference software).

3.4 Results of coding with JM reference software

This section will discuss the resulting noise and artifacts that occur when encoding/decoding both artificial content and natural video. For this experiment, we use a QP of 40 for all frames (I and P-frames) to ensure a low bit rate and thus ensure the occurrence of noise artifacts. In addition, the parameters are set such that we use an IntraPeriod of eight (i.e., the GOP-size is eight) and no B-frames ($NumberBFrames = 0$), to reduce the coding duration. 50 frames of the sequence were coded. The resulting bit rate is 275.56 kbps with an average PSNR of 29.361 dB for the Y frames. The video used was 'screen_content_1' as the sample artificial content video.

Figure 3.5 shows a comparison between a still image from the original screen_content_1 sequence (left image) and a still image from the sequence coded with the JM reference software (right image). We see from the right image that the text has been noticeably blurred. In addition to this, there are ringing artifacts surrounding the text. This is especially noticeable in

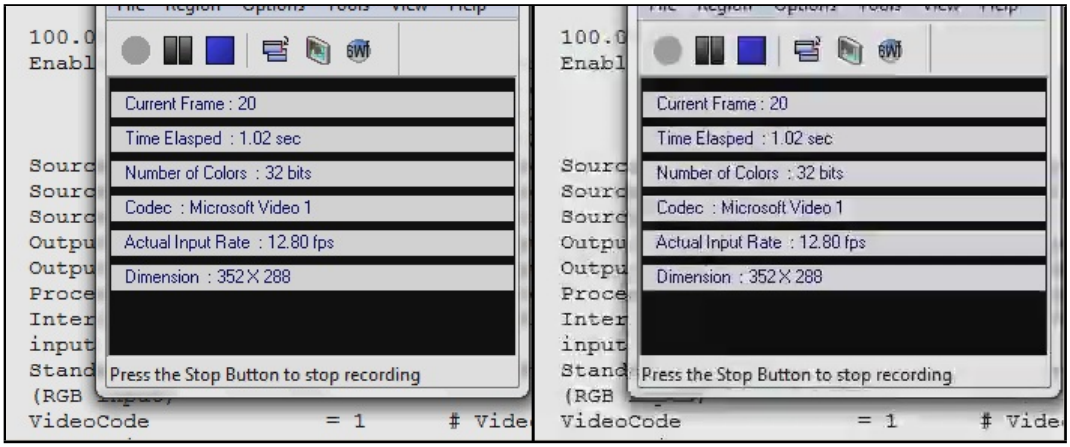

Figure 3.5: A comparison between the original (left) and reconstructed (right) screen_content_1. A QP of 40 for all frames was used. An enlarged version of the image may be found in Appendix C.

the frame containing the CamStudio parameters. The ringing artifacts and the blurred text are both caused by the component transform performed by the coder before quantization. When quantizing these transformed components using a uniform quantizer, it is analogous to truncating the transformed signal, i.e. reducing the accuracy of the approximation of the transform, resulting in Gibbs phenomenon[30]. This can also be thought of as filtering the signal with a low-pass filter. As mentioned, Gibbs phenomenon occurs when truncating the series expansion of the signal, which is equivalent to reducing the value of K as shown in Figure 3.6, resulting in increased oscillations around the points of discontinuity and consequently visible ringing artifacts. The transform used in Figure 3.6 is the Fourier transform, and not the integer transform used in H.264. However, the integer transform is similar to the Fourier transform in that the original signal is represented by a sum of basis functions. It consequently holds true for any transform that reducing the resolution will result in reduced accuracy for the truncated signal.

As the human visual system is less sensitive to high-frequency components, this works well for natural video, which tends to contain mostly low-frequency components due to the isotropic property. However, as CGC normally contains a large amount of high frequency components, i.e. sharp edges, the standard coding process might not perform optimally when compressing artificial content. The coding algorithm should therefore be modified to take this into account. A possible modification is proposed in Chapter 4.

Figure 3.7 shows an example of natural video being coded also using

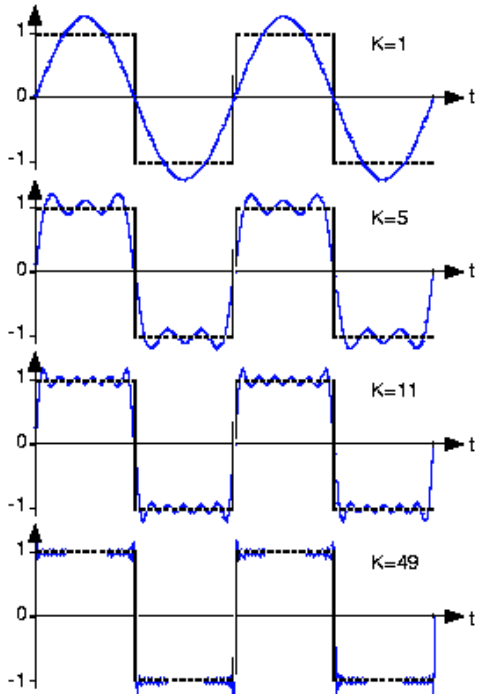

Figure 3.6: Illustration of convergence of the Fourier transform and the Gibbs phenomenon at the point of discontinuity. Taken from [31].

Figure 3.7: A comparison of original .yuv file (left) and coded .yuv file (right) using a QP of 40. 'foreman' sequence.

a quantization parameter of 40. The rest of the also parameters remain unchanged. The resulting bit rate is 119.36 kbps, which is less than half

Video	**Mean PSNR**	**Bit rate (kbps)**
foreman	30.644	119.36
screen_content_1	29.361	275.56

Table 3.1: A comparison of the results obtained when encoding natural video and artificial content using the JM reference software.

of the bit rate when coding the artificial content video, and the average PSNR for Y frames was 30.644, which is slightly better than the artificial content, as shown in Table 3.1. We thus see that for these two videos, the JM reference software does a better job coding natural video compared to artificial content, with a slightly larger PSNR for less than half the bit rate. However, as discussed in Section 2.4, PSNR does not necessarily work well as a visual quality metric across different source content, like natural video and screen content. Still, it does give an indication of what to expect when using H.264 to code the various types of content.

The results of the conducted experiments thus agrees with the statements made in the beginning of the thesis, i.e. the sub-optimality in terms of visual quality when coding artificial content using the H.264 coding algorithm.

Chapter 4

Introducing Residual Scalar Quantization

In this chapter we will introduce a method that may improve upon the already existing techniques, by taking the characteristics of artificial content into consideration when performing the compression.

4.1 Investigating RSQ

The first method investigated in the thesis is *residual scalar quantization* (RSQ)[17][1]. The method is divided into two main parts. The first part is modifying the intra-prediction scheme to better reflect the characteristics of artificial content. We assume that the shorter the prediction distance is, the stronger the correlation between the pixels will be. Therefore, it uses pixel by pixel prediction, i.e. it uses each pixel within a block to predict the neighboring pixel in direction given by the intra prediction mode. This scheme is depicted in the right part of Figure 4.1, which shows a slightly modified version of mode '4' in the current H.264 intra-prediction scheme (the standard H.264 intra-prediction modes can be seen in Figure 2.5). In the proposed scheme, pixel 'b' will be predicted using pixel 'a' instead of the reconstructed sample 'M', as would have been used in the standard H.264 algorithm. The left part of Figure 4.1 shows the eight possible prediction directions, in addition to mode '2' which is the *DC* mode, where the prediction

[1]The *scalar* in residual scalar quantization refers to the type of quantization used. Scalar quantization is the quantization we have used in this thesis, where each coefficient is quantized separately. The alternative method is called *vector quantization*, where coefficients are grouped together and then quantized, allowing for increased compression rates at the cost of an increase in complexity.

values are averaged from adjacent samples. These directions are equivalent to the prediction directions used in standard H.264.

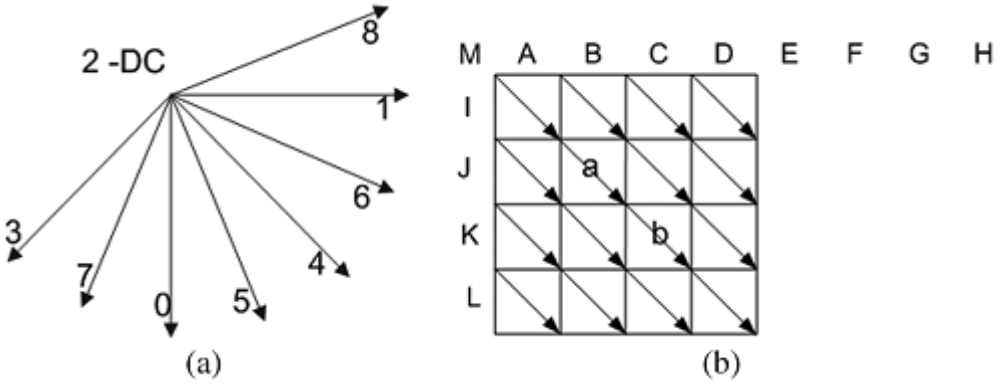

Figure 4.1: Intra prediction in RSQ mode. Taken from [17]

.

As previously mentioned, the H.264 algorithm performs a component transform on the residual signal after performing intra- and inter-prediction. This is followed by a quantization of the transformed components. However, as described in Section 2.3, performing a component transform of artificial content may actually make the signal less compact, which is undesirable. Therefore, assuming that the signal is sparser in spatial domain than in the transform domain, we introduce the second and most important part of the technique. In contrast to most of the earlier video coding algorithms, we propose to omit the component transform, and instead quantize the residual signal directly.

4.2 Implementing the RSQ coder

A version of the H.264 coder has been written in *MATLAB* to compare the techniques currently used in H.264 to the proposed techniques. The coder is significantly simplified compared to H.264, but the coding process still uses the same key concepts such as prediction, transform, quantization, and entropy coding. Entropy coding will be performed to give us the ability to measure the amount of bits required to represent the coded information.

The following open source libraries were used in the two coders:

- **YUV Toolbox** - For converting between color spaces, and reading/writing .yuv files.

- **Huffman** - To perform Huffman coding, which will function as our simplified version of entropy coding.

A further description of these tools may be found in Appendix A.

We use the 2D DCT for the transform coding, as the integer coding used in H.264 is an approximation to DCT and MATLAB has a built-in 2D DCT function called *dct2()*. The transform is done in a block-wise manner using 4x4 blocks, and the resulting transformed blocks are then quantized using uniform quantization. After quantization, the coefficients are Huffman coded to calculate the total amount of bits used with variable length coding[2]. The entropy coding is done on a frame-by-frame basis instead of on a macroblock basis, as it is done only to indicate the amount of bits used to represent the coded sequence. We will use the term *traditional coder* when referring to this coder.

A second coder was also created with the purpose of testing the proposed RSQ technique. This coder consequently omits the component transform and directly quantizes the signal using a uniform quantizer, effectively reducing the value range of the output signal. This coder will be referred to as the *RSQ coder*. The uniform quantization for both coders is done as follows:

$$Q(I) = \left\lfloor \frac{I}{QP} \right\rceil,$$

where QP denotes the given quantization parameter, I denotes the input signal and $Q(I)$ denotes the quantized output signal. The reconstruction process is given by:

$$\tilde{I} = Q(I) \times QP,$$

where $\tilde{I}$ denotes the reconstructed signal.

Both coders take in an .avi file which is first read to RGB and then converted to YUV. This is a lossless operation shown in Equation 2.1. The entropy coding is done by passing a frame or block to the *norm2huff()* function from the *Huffman toolbox*[34]. The input to the *norm2huff()* function must be an *unsigned 8-bit integer vector*, which puts some constrains on the input signal. Consequently, we add the absolute value of the minimum pixel value in the block or frame to all coefficients to ensure that the minimum value of the input signal is zero. The maximum possible value will thus be $Max = \frac{255}{QP} \cdot 2$, which will yield maximum values ≤ 255 for all $QP \geq 2$, ensuring that the input values are within the valid range. The *norm2huff()*

[2]Variable length coding refers to the use of variable bit code lengths when assigning codewords to sample values. The alternative is to use fixed-length coding, where all sample values are assigned equal length codewords.

function then calculates the Huffman codewords, which may be used to generate a list of prefix-free bit codes. The most probable values in the input block are then mapped to the shortest bit codes, as explained in [8]. An *info* structure is returned for each call to *norm2huff()* which holds information about the following:

- **Pad** - Holds information about how many bits that have been added at the end of each bit sequence
- **Huffcodes** - This contains the Huffman codewords and is the parameter that must be passed to the *huffcodes2bin()* function to get the list of Huffman bit codes
- **Ratio** - The compression ratio
- **Length** - Original data length
- **Maxcodelen** - Maximum codeword length

The code snippet below shows how the entropy coding process is done in our simplified coders. The *codedYFrames* matrix contain the quantized coefficients of the coded video sequence.

```
% Entropy coding
for i = 1:nFrames
  % The entropy coder only works for uint8, so we add the
  % lowest value to ensure that all values are >= 0.
  tempValue(i) = abs(min(min(codedYFrames(:,:,i))));
  tempBlock = codedYFrames(:,:,i) + tempValue(i);
  % Entropy code frames
  [tempCell, info] = norm2huff(uint8(tempBlock(:)));
  [k,l,huff] = find(info.huffcodes);
  % Create bitcodes from huffcodes
  listOfBitCodes = huffcodes2bin(info.huffcodes);
  % Sort frequency of codes to match huffman bit codes
  % (most frequent = shortest bit codes)
  sortedCodes = zeros(1,length(huff));
  for j = 1:length(huff)
    sortedCodes(j) = sum(sum(tempBlock==huff(j)));
  end
  sortedCodes = sort(sortedCodes,'descend');
  bitsPerFrame = 0;
  tmpBits = 0;
  % Find total bits used by multiplying the length of
  % the bit codes with the frequency of the codeword,
  % then add the value of the coefficient in the translation
  % table.
  for j = 1:length(huff)
```

```
26      tmpBits = length(listOfBitCodes{j})*sortedCodes(j);
27      bitsPerFrame = bitsPerFrame+tmpBits+ceil(log2(huff(j)));
28    end
29    totalBits = totalBits + bitsPerFrame;
30    infoVector(:,i) = struct2cell(info);
31    huffmanCoded(i) = {tempCell};
32  end
```

We see that the number of bits used by each coder are calculated by sorting the amount of values in each input-block by descending frequency of occurrence in the *sortedCodes* vector (i.e. the most probable value is set to be the first element in the list, the second most probable is the second element etc.). The frequency of occurrence for each pixel value is then multiplied by its corresponding codeword length (i.e. the length of the corresponding *listOfBitcodes* component), which we get from passing the *huffcodes* field from the *info* structure to the *huffcodes2bin()* function. Lastly, we add a bit representation of the sample value ($\lceil \log_2 x \rceil$, where x denotes the pixel-value). This is not sufficient for a real working coder, as we have not even considered the indices of the values, thus making reconstruction impossible. Unfortunately the limitations of the Huffman coder which was mentioned earlier makes it difficult to implement the coding of indices efficiently, and we will hence continue using this simplified approach for this experiment. It will most likely not affect the end result largely as we use the same approach for both coders in the comparison. The number of bits for each frame is then added together to find the total number of bits used for the entire video.

The inverse entropy coding is shown in the code snippet below.

```
1  % The dim variable is chosen such that
2  % size(infoVector(:,i),dim) == length(fields)
3  dim = 1;
4  fields = fieldnames(info);
5  % Inverse entropy coding
6  for i = 1:nFrames
7    tempVector = cell2mat(huffmanCoded(i));
8    info = cell2struct(infoVector(:,i),fields,dim);
9    tempBlock(:) = huff2norm(tempVector,info);
10   codedYFrames(:,:,i) = tempBlock - tempValue(i);
11 end
```

We see that the Huffman codewords returned from the *norm2huff()* function are passed to the *huff2norm()* function along with its corresponding info structure *info*. If the input is kept within the accepted boundaries (unsigned 8 bit integers), the reconstruction of the Huffman coded input is done without loss of information.

Both coders then perform inverse quantization and inverse differential

Video	**Traditional coder**		**RSQ coder**	
	Mean PSNR	Bits used	Mean PSNR	Bits used
foreman_cif	33.4392	9936359	29.0175	19145420
screen_content_1	33.329	14402185	30.2489	10761434
screen_content_2	32.1377	96858066	29.3447	76279621

Table 4.1: Comparison between traditional coder and RSQ coder. QP 32, no prediction.

coding on the frames resulting in a reconstructed compressed series of images. These are then converted to a yuv-video by using the *yuv_export()* function from the *YUV toolbox*[39]. The PSNR of the reconstructed video is then calculated to give an indication of the visual quality of the reconstructed images as well as for comparison purposes.

The full MATLAB code for the coders used in this chapter can be found attached under the names 'trad_coder_no_pred.m' and 'rsq_coder_no_pred.m'.

4.3 Results from coding with RSQ

The screen content was coded using both the traditional coder and the RSQ coder, with a uniform quantizer with a QP of 32 for both coders. The tests were conducted using all three example videos presented in Section 3.2. We only focus on the Y-frames (i.e. we ignore the U and V frames) in all experiments for the sake of simplicity. To isolate the issue of using a component transform versus not using a component transform before quantization, we avoid using any form of intra- or inter-prediction for this particular experiment.

Table 4.1 contains the coding results for each sample sequence. The results of the coding is measured in terms of total bits used and objective visual quality (PSNR). It may be difficult to see how one coder compares against the other just by looking at the PSNR and total amount of bits used. To make the comparison of the results more clear, we will calculate the Lagrange number given by Equation 2.5 as a metric for comparison. We will use $\lambda = 0.85$ throughout the rest of this thesis as suggested in [18]. We may now compare J for each coder, and the coder that achieves the minimum J is the most efficient coder for this particular scenario. We calculate the distortion inversely from the PSNR, which translates to Equation 4.1, where

Video	Traditional coder	RSQ coder
foreman_cif	11430000	24530000
screen_content_1	15300000	15370000
screen_content_2	86350000	72500000

Table 4.2: Comparison; Traditional coder and RSQ coder, no prediction. The values from Table 4.1 have been converted to Lagrange values for an easier comparison.

M and N denotes the height and width of the image.

$$D = \frac{255^2 \times M \times N}{10^{\frac{PSNR}{10}}} \tag{4.1}$$

Table 4.2 shows the resulting Lagrange numbers for each coder. We see that the traditional coder slightly outperforms the RSQ coder when coding the video 'screen_content_1'.

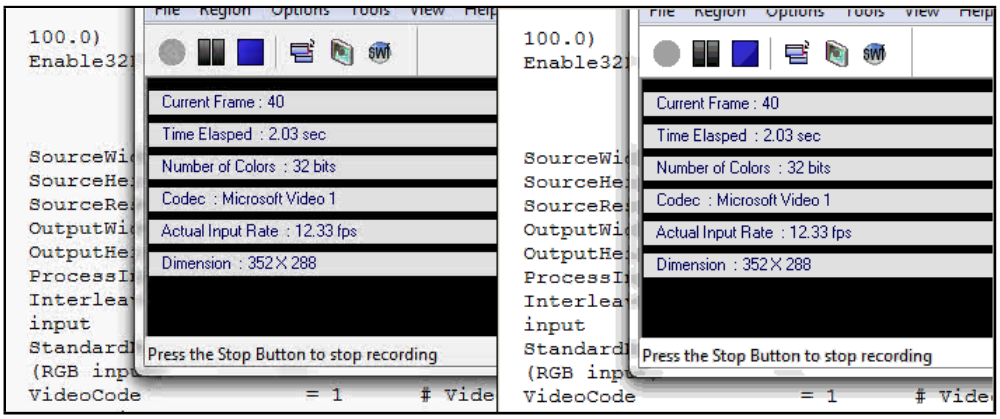

Figure 4.2: A comparison of the two coders, traditional (left) and RSQ (right). QP 32, no prediction was used. Source signal is screen_content_1. An enlarged version of the image may be found in Appendix C.

However, Figure 4.2 shows that the right image, which is taken from the RSQ coded sequence, is significantly less noisy than the left image which is taken from the traditionally coded sequence. The left image contains ringing artifacts around areas containing text and graphics, whereas the RSQ coder

contains few noticeable noise-artifacts. The only noticeable difference is a shift in grayscale for some areas, i.e. some areas turn a darker shade of gray, whereas some areas turn a lighter shade of gray which is caused by the reduced size of the range of possible output pixel values.

Because the traditional coder achieves such a significantly larger PSNR as shown in Table 4.1, but is still affected so severely with noise and artifacts, we may conclude that the PSNR not necessarily correlates well with subjective visual quality, even when coding the same source content. The RSQ coder also uses fewer total bits, though this might not be a fair comparison given the simplified environment of the entropy coders which are introduced to measure the rate.

Looking at the results when coding 'screen_content_2' in Table 4.2, we see that the resulting Lagrange numbers are smaller for the RSQ coder compared to the traditional coder, meaning that the RSQ coder does a better job at coding the content of this video. As we saw earlier, the PSNR reported is not necessarily consistent with the actual achieved visual quality for the reconstructed video sequence. Comparing the two videos 'traditional_scc2.yuv' and 'rsq_scc2.yuv' which are attached together with the thesis, we see that the traditionally coded video suffer from the noise artifacts we have previously identified, while the RSQ coded video contains few artifacts but still achieves a significantly lower PSNR, as can be seen in Table 4.1.

Based on the results we have seen in this section, we might conclude that the RSQ coder indeed works well when coding our example screen content videos. However, the simplified nature of the coders we have implemented make it difficult to verify the accurateness of the comparison. To ensure a fully satisfactory comparison, the optimal process would be to implement RSQ in an already existing H.264 implementation such as the JM reference software. This would give us a more complete idea of how the technique interacts with the other components of H.264, such as inter- and inter-prediction, quantization and entropy coding. Unfortunately, the source code for the JM reference software proved to be too complex to effectively implement the RSQ coder within the alloted time slot. Therefore, an alternative and simplified method was chosen. Still, the accumulated results from the simplified coders may still give an indication of how the proposed techniques will perform when implemented in a full-scale H.264 implementation.

4.4 Problems with the RSQ coder

The RSQ coder is tailored for artificial content. We may therefore have reason to suspect that it will not work as well when used together with

natural video. Figure 4.3 shows the *'foreman'* sequence coded by using the traditional coder (left) and the RSQ coder (right). The PSNR and total bits used may be found in Table 4.1. Indeed, we see that the traditional coder significantly outperforms the RSQ coder, both in terms of bits used, PSNR and actual visual quality. This is as expected and agrees with the theory we have discussed earlier.

Figure 4.3: Comparison of the two custom coders using a natural video source. The left image is coded using the traditional coder. The right image is coded using the RSQ coder.

As the RSQ coder clearly does a sub-optimal job when coding natural content, it might present a problem if the CGC that is to be coded contains natural images, i.e. compound content, as depicted in Figure 4.4. The computer graphics are hardly subject to noise and artifacts, but the embedded real image is clearly distorted. A possible solution is to implement an adaptive coder, which selects the best possible coding option based on the information or source that is to be coded. We will look further into this in Chapter 5.

The reason the RSQ technique does not work well with natural video is the reduced color resolution which was mentioned in the previous section. As each pixel is represented by eight bits of luma information (in addition to 16 bits of chroma information, which we do not consider in this thesis), the pixel values lie in the range $[0, 255]$. When we use uniform quantization and simply divide by a given quantization parameter, as is done in our custom coders, it is equivalent to reducing the size of the dynamic range of the output values from $[0,2^8]$ to $\left[0, \left\lfloor \frac{2^8}{QP} \right\rfloor\right]$. The result is that the RSQ coder will not work well for frames with a large variety of colors and color nuances, which is typical for natural video, but also for some artificial content. It will however

Figure 4.4: Computer graphics with embedded natural image coded using the RSQ coder from the screen_content_2 video.

excel when coding areas with few colors and many graphical objects, such as large areas with text. Our simplified quantization scheme is also sub-optimal in the sense that using a potentially non-dyadic[3] maximum value for the dynamic range, as we need $\left\lceil \log_2 \frac{2^8}{QP} \right\rceil$ bits to represent the value anyway. If the QP value is non-dyadic, the result is a waste of information as we use x bits to represent values that do not fill the range from 0 to 2^x completely. A method to improve upon this is investigated in Chapter 6.

[3] A dyadic number is defined as 2 to the power of a natural number b (i.e. 2^b).

Chapter 5

Introducing Adaptivity

In this chapter we will introduce adaptivity to enhance coding results by allowing the coders to select the best performing coding technique for each block in a frame.

5.1 Implementing the adaptive coder

There may be several scenarios when the RSQ method will not be the optimal choice. Such scenarios may include natural images or video embedded in artificial content, which we have defined as compound content. But it may also be the case that for some instances of artificial content, the traditional coder will outperform the RSQ coder. We may thus use RDO, which was introduced in Section 2.5, to ensure that the optimal coder is used for each coded block. To use RDO, we need to calculate the distortion and bit rate for each block. These values are then weighted and compared as previously discussed.

The distortion is already available to us, as we just calculate the distortion between the original block and the reconstructed block as described in Equation 2.6. However, to calculate the rate, the coder has to entropy code each block separately, so the coder must be modified to entropy code on a block-wise basis instead of a frame-wise basis. To be able to do this correctly, each frame is split up into a three dimensional matrix of size $[b, b, (M \times N)/b^2]$, where b denotes the block size (four in our experiment) and M and N denotes the height and width of the frame to be coded. Each block is then processed independently by each of the two coders. The reconstructed frames are created by optimally choosing the best coded block from each coder using RDO optimization.

As previously mentioned, the entropy coding method is severely con-

strained and only accepts inputs of 8-bit unsigned integer format. Hence, it does not work as intended when the input blocks contain values that are located outside of this interval. Moreover, blocks that are entirely uniform in value will result in the *norm2huff()* function breaking down and returning a value of zero. Therefore, if the content of the block is uniform, the coder must store the actual value and transmit it as side-information so that it may be added to the block of zero-valued coefficients that results from passing the zero returned from the *norm2huff()* to the *huff2norm()* function.

Another issue we encounter is when we code screen content such as in the *screen_content_2* video, the temporal redundancy is so large that when we use differential encoding on two consecutive frames, the latter residual frame will often consist of only zero-valued coefficients. As the entropy coder cannot process a frame consisting of all zeros, we have in these cases assigned each block one bit, for the sake of being able to properly compare the two coders. In this coder we have also introduced prediction. As the purpose of this experiment is to maximize the coding output and not directly comparing the two coders, we have implemented different prediction schemes for both coders. The reason for choosing different prediction schemes and only using one of the two prediction schemes for each coder is that using both prediction schemes on the same coder causes our non-robust coder to break down. We therefore choose to use the type of prediction that maximizes the PSNR for each coder. This was found to be differential frame-wise inter-prediction for the traditional coder. In this differential coding scheme, we subtract the previous I-frame (anchor frame) from each following P-frame to reduce the size of coefficients by exploiting the large temporal redundancy occurring in artificial content. It follows an *open-loop* structure, i.e. the P-frames are only predicted from the previous I-frame. However, this is sub-optimal in terms of prediction because of the increasing distance between the predicted frame and the anchor (I) frame. The optimal solution would be to predict each P-frame using the preceding P-frame, a *closed-loop* structure. However, because the closed-loop structure results in increasing additive noise for each consecutive P-frame, only resetting at every I-frame, we were forced to sacrifice the gain in bit rate and use a open-loop structure instead for our simplified coder.

For the RSQ coder, we use a simplified version of the neighboring pixel intra-prediction scheme discussed in Section 4.1. Our version of the intra-prediction scheme uses column-wise differential coding, i.e. the top-left pixel of each 4x4 block is used to differentiate all other pixels in the block in a column-wise matter. The residual pixel values are then uniformly quantized using a predetermined quantization parameter (QP). We will use an I-frame frequency of six, i.e. the GOP-size is six. λ will be set to 0.85 according to [18].

The adaptive coder used in this section may be found attached under the name 'adpt_coder_intra_pred_rsq.m'.

5.2 Results of adaptive coding

The adaptive coder codes each frame block-by-block using both the traditional coder and the RSQ coder, as shown in Figure 5.1.

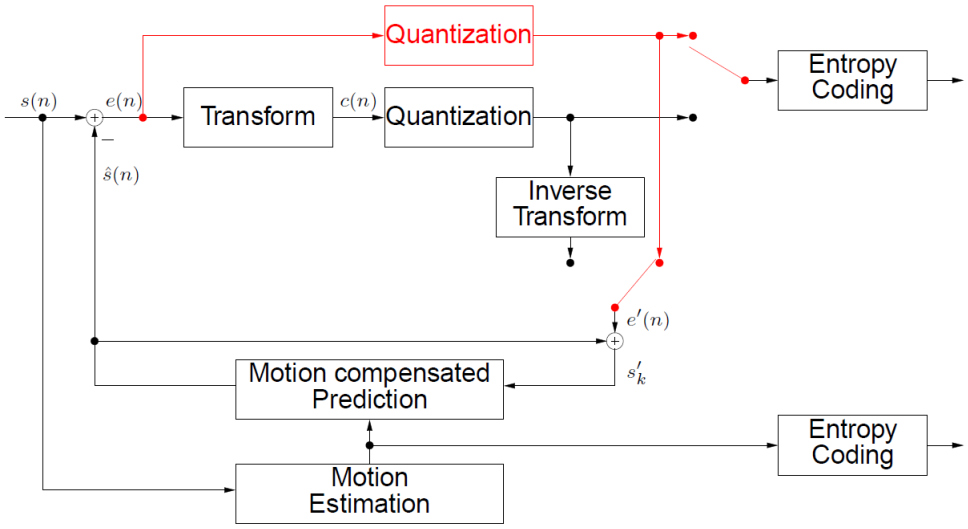

Figure 5.1: Block diagram of the RSQ coder. Taken from [27]

It then uses rate-distortion optimization to find the most effectively coded block, in terms of bit rate (in this coder the bit rate refers to the amount of bits per block) and distortion. The result is a coder that outperforms both of the standalone coders in terms of PSNR as seen in Table 5.1. The bit rate will be slightly higher than the minimum bit rate for each frame, but the gain in PSNR outweighs the slight increase in bit rate.

The experiment was conducted using all three videos listed in Section 3.2. A uniform quantizer with a quantization parameter of 25 was used for all the tests.

Figure 5.2, Figure 5.3, and Figure 5.4 shows three coded images from the sequence 'screen_content_1' coded by the adaptive coder. Figure 5.2 is coded with the traditional coder. We see that it contains artifacts in the shape of blurred text and ringing artifacts around text and graphical objects. Figure 5.3 is coded by the RSQ coder. This image contains some discolored

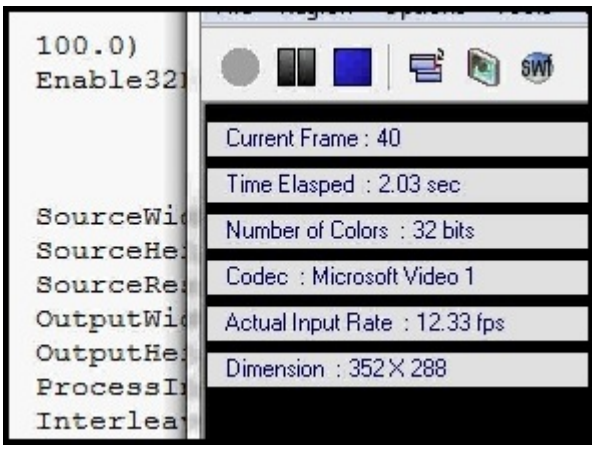

Figure 5.2: 'screen_content_1' video coded using the traditional coder. A QP of 25 was used.

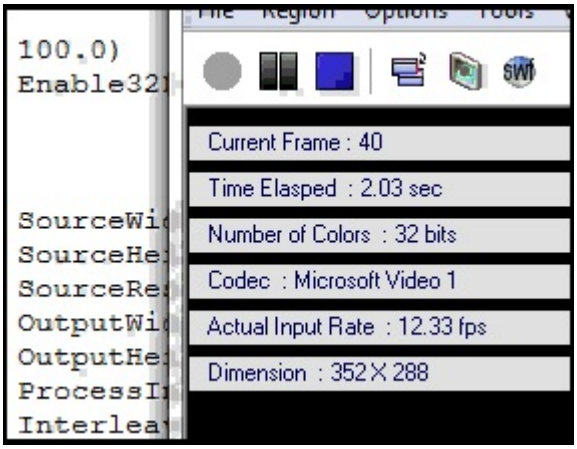

Figure 5.3: 'screen_content_1' video coded using the RSQ coder. A QP of 25 was used.

blocks from the intra-prediction and some of the graphical objects on top of the screen suffers from a reduced grayscale resolution. Figure 5.4 is the adaptively coded still image. This frame is composed of the optimally coded blocks from the traditional coder and the RSQ coder. This image contains less noise and artifacts than the two images on the left, which is as expected and it shows that our adaptive coder works as intended.

Similarly to what we did in Section 4.3, we use the information in Table 5.1 to calculate the Lagrange value for each coder and each video for an easier

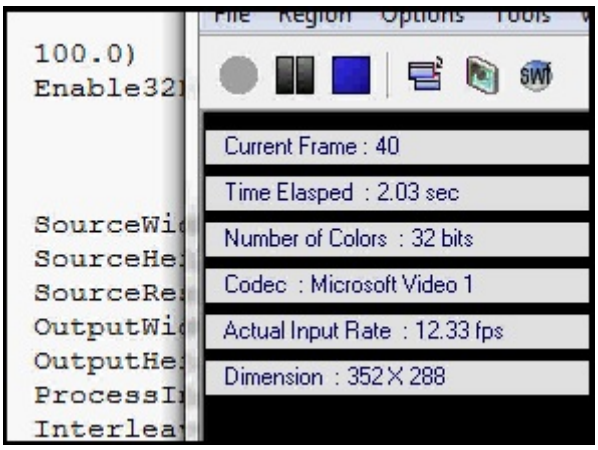

Figure 5.4: Still image 'screen_content_1' video coded using the adaptive coder. The resulting video is composed of the best coded blocks from each coder. A QP of 25 was used.

Video	**Traditional**		**RSQ**		**Adaptive**	
	Mean PSNR	Total bits	Mean PSNR	Total bits	Mean PSNR	Total bits
foreman_cif	32.4383	12374972	29.0010	14734351	32.6751	12278923
scc1	36.2524	4070450	34.8708	2548396	37.4173	2294093
scc2	35.3048	12815549	32.8768	21071180	35.7584	11312582

Table 5.1: Results of encoding using the adaptive coder and a QP of 25 for all three test video sequences. *scc1* and *scc2* are abbreviations for 'screen_content_1' and 'screen_content_2' respectively.

comparison of the total performance of the coders in terms of visual quality and bit rate. The results can be seen in Table 5.2. The best performing coder is accordingly the coder that achieves the lowest Lagrange value.

We see from Table 5.2 that the traditional coder outperforms the RSQ coder in terms of achieving the lowest Lagrange value (J) when coding natural video ('foreman') and when coding compound content ('screen_content_2'). As the traditional coder was originally intended to code natural video, it comes as no surprise that the RSQ coder is outperformed when coding the 'foreman' sequence. However, the results from coding the 'screen_content_2' are not as easily explained, as the traditional coder outperforms the RSQ coder. There may be several reasons for this. First, large areas of the source video consists of natural images, which we have shown are not optimal for

Video	Traditional	RSQ	Adaptive
foreman_cif	14279000	20821000	13998000
screen_content_1	5022200	4313600	3144800
screen_content_2	17518000	29498000	15583000

Table 5.2: Lagrange optimization values for each coder derived from the results in Table 5.1. $\lambda = 0.85$

the RSQ coder. Secondly, this video contains little motion between frames except for the abrupt transitions when switching from on web page to another, as well as the opening of the PDF file towards the end of the movie. Because of the large temporal redundancy between neighboring frames, the traditional coder that uses differential coding between frames is well suited and will use a small amount of bits to represent the image residuals (which will consist of a lot of zero coefficients).

We also see from Table 5.2 that the RSQ coder outperforms the traditional coder in terms of achieving the lowest Lagrange value when coding the video 'screen_content_1'. As 'screen_content_1' consists solely of artificial content, these results support the theory discussed in previous sections, and it indicates that a component transforms should not be used when coding artificial content.

Even if the RSQ coder does not perform optimally under all given circumstances (most noticeably when coding natural video), we see that the adaptive coder outperforms both of the standalone coders, both in terms of PSNR and total bits used, independent of video content. As the purpose of the experiment was to see if the adaptive coder would really outperform the standalone coder, we conclude that the implementation was successful.

This implies that if the duration of the encoding process is not an issue, the adaptive coder should always be chosen. If the encoding is to be done in real-time however, the issue becomes more complex, as the encoding process may take too long. However, this is outside the scope of the thesis.

5.3 Problems with the adaptive coder

This section will briefly touch upon an important weakness of the adaptive coder. As seen in Section 4.3, the reconstructed frame from the RSQ coder reports a significantly lower PSNR than when coding the same artificial content using the traditional coder even when the RSQ coder contains less noise artifacts. This clearly presents a problem when the RDO method is based

on the same metric (the square error) as the PSNR, when the optimal block may actually be the block achieving the largest square error. We will touch upon this topic again in Chapter 7.

Chapter 6

A Closer Look At Quantization

In this chapter, we will look at the uniform quantization used in the coders we have used throughout this thesis (excluding the JM reference software) and we attempt to improve upon the quantization by considering the characteristics of the source signal to be coded.

6.1 Improving the quantizer

In the experiments conducted in the previous two chapters, we have used a simplified uniform quantizer which divides all values in a block by a predetermined quantization value as explained in Section 4.2.

To improve upon the already existing quantizer, we begin by implementing a slightly more advanced, but still uniform quantizer. This quantizer divides the dynamic range of the signal into N representation levels, where $N = 2^B$ and B represents the amount of bits used in the quantizer. This forces a dyadic amount of representation levels and thus a more effective bit-wise representation compared to our previous quantization method, as discussed in Section 4.4. The distance between each representation level (quantization step size), denoted Δ, is defined as $\Delta = \frac{\text{Max–Min}}{N}$. Min represents the minimum value of the block or frame and Max represents the maximum value (for 24 bit images, i.e. eight bit per color component, Min and Max are usually 0 and 255 respectively). The uniform quantizer is given by Equation 6.1.

$$Q(y) = \text{sign}(y) \cdot \Delta \cdot \left(\left\lfloor \frac{|y|}{\Delta} \right\rfloor + \frac{1}{2} \right) \tag{6.1}$$

Here, y represents the input coefficient and $Q(y)$ represents the quantized coefficient. As mentioned, the uniform quantizer is simplified in that it does not consider the characteristics of the source signal. In our case the source

signal will consist of mostly artificial content, which possesses some unique characteristics as discussed in Section 2.3. One important characteristic is that artificial content often contains areas and objects that are either white (pixel value 255 or close to it) or black (pixel value zero or close to it). An example is black text on white background, which is a common scenario when referring to artificial content. Considering this, we might want to modify our quantizer to reflect this observation, as shown in the equation:

$$Q(y) = \begin{cases} \text{Min}, & (y < \text{Min} + \frac{3}{2} \cdot \Delta) \\ \text{Max}, & (y > \text{Max} - \frac{3}{2} \cdot \Delta) \\ \text{sign}(y) \cdot \Delta \cdot \left(\left\lfloor \frac{|y|}{\Delta} \right\rfloor + \frac{1}{2} \right), & (\text{Min} + \frac{3}{2} \cdot \Delta \leq y \leq \text{Max} - \frac{3}{2} \cdot \Delta) \end{cases}$$

Assuming Min and Max will normally be 0 and 255 (or close), the values that would usually be mapped to $Min + \frac{\Delta}{2}$ or $Max - \frac{\Delta}{2}$ according to Equation 6.1, will now be mapped to Min and Max instead, which are more probable values in artificial content.

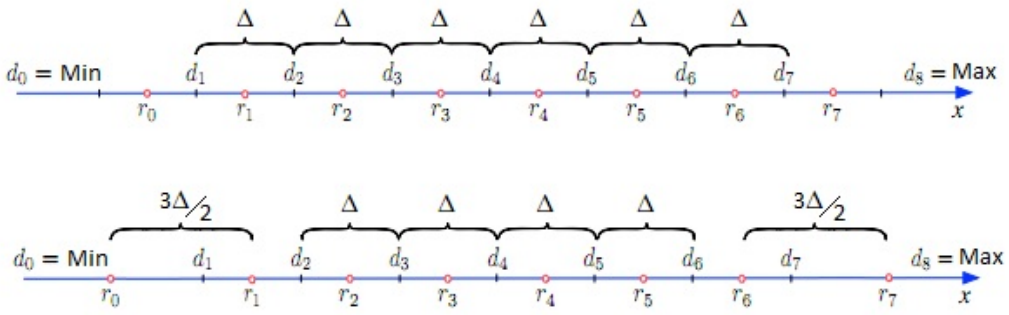

Figure 6.1: Uniform quantizer (top). Quantizer with extended end-zones (bottom). The top image is taken from [5].

Figure 6.1 shows how the two different quantizers are divided with respect to decision values (d_i) and representation values (r_j). Both quantizers are three bit, and we see that the only difference is the displacement of r_0 and r_7, instead of being regularly spaced with a distance of Δ from r_1 and r_6, they are now placed at the Min and Max values. All values that are smaller than d_1 and larger than d_7 will hence be mapped to the output values r_0 (Min) and r_7 (Max) respectively. All values in between d_1 and d_7 will be quantized uniformly.

This extended end-zone quantizer will not be implemented for the traditional coder. The reason for this is that mapping frequency domain values

(which is the values that are quantized in the traditional coder) to their maximum and minimum values do not grant any particular benefits, as the values in the frequency domain does not directly translate to their respective values in the spatial domain.

6.1.1 Comparison between regular and proposed quantization

Figure 6.2 shows a comparison between the uniform quantizer and the quantizer with extended end-zones using the video 'screen_content_1' as source signal. We have compared the results when inputting one through seven bits in the quantizer and we will compare the resulting PSNR for each coder. As the amount of representation levels are the same for both coders, we will assume equal bit rates. All parameters are equal for both coders except the modified boundary representation levels in the extended end-zone quantizer.

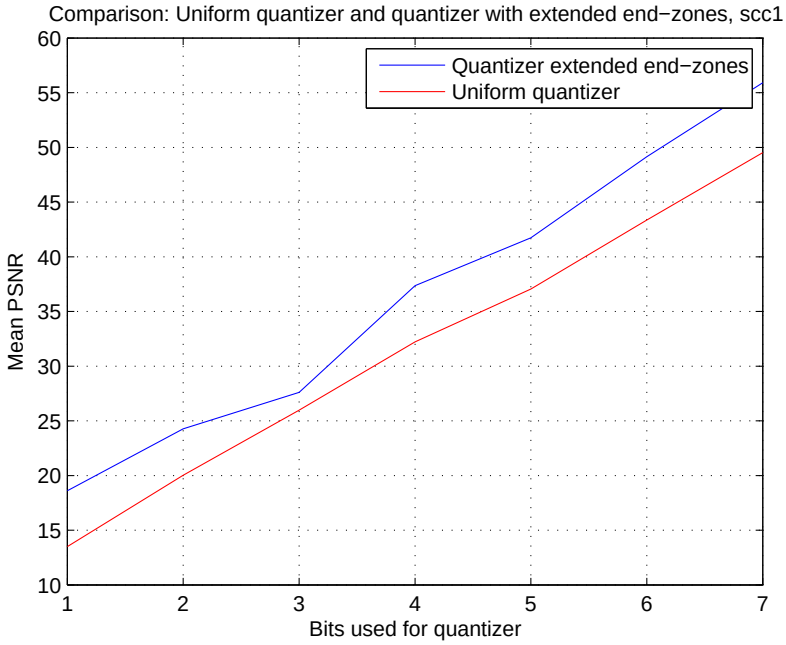

Figure 6.2: A comparison between the uniform quantizer (red line) and the quantizer with extended end-zones (blue line). The source signal used is the 'screen_content_1' sequence.

Figure 6.3 shows a comparison between the two quantizers using the video 'screen_content_2' as source signal.

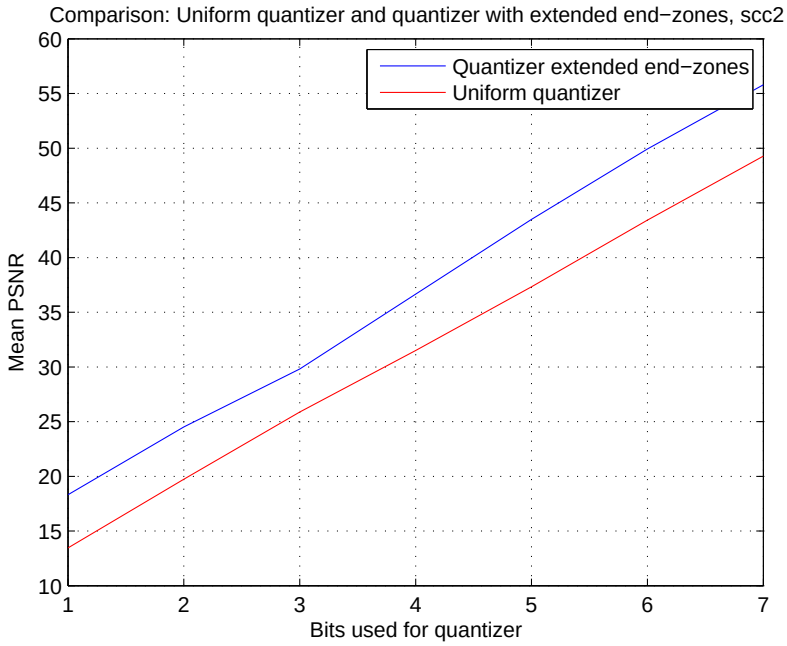

Figure 6.3: A comparison between the uniform quantizer (red line) and the quantizer with extended end-zones (blue line). The source signal used is the 'screen_content_2' sequence.

As we see, the uniform quantizer with end-zones at Min and Max greatly outperforms the regular uniform quantizer for all amount of bits used in the quantization process, with the largest difference being 6.5273 dB for the seven bit quantizer when coding 'screen_content_2' and 6.3949 dB for 'screen_content_1', which is a significant difference. The uniform quantizer with extended end-zones should thus be chosen over regular uniform quantization as long as the source content is artificial or compound with mostly artificial content.

6.2 Initial RSQ or end-zone RSQ

In this section we will attempt to compare the RSQ coder that was introduced in Chapter 4 (which we will henceforth denote the *initial RSQ coder*) and the RSQ coder using the quantizer with extended end-zones (which we will henceforth denote as the *end-zone RSQ coder*).

Unfortunately, when implementing the quantizer with extended end-zones

in the RSQ coder, the entropy coder which is used to calculate amount of bits used does not work properly, resulting in the coder reporting an unrealistically low amount of bits used. The reason for the low amount of bits reported is that when passing the coded frame to the entropy coder, it will seemingly randomly add ± 1 to each coefficient of the vector *huff*, which contains all original values in the block or frame to be coded. Hence, when we want to learn how many appearances there are of each value from *huff* in the input block to the *norm2huff()* function (i.e. *tempBlock*), as shown in the code snippet below, it will report zero for each value, in turn leading to the break down of our bit rate measurement.

```
1 for j = 1:length(huff)
2   sortedCodes(j) = sum(sum(tempBlock==huff(j)));
3 end
```

The reason for the seemingly randomly addition of ± 1 is unknown. We will continue with the comparisons, but the bit rate will not be considered for those experiments where the coder fails to correctly calculate the amount of bits used. This makes a comparison between the initial RSQ coder and the end-zone RSQ coder more difficult as we may only consider the PSNR. However, we will assume that the dynamic range of the output values are approximately the same, resulting in the actual bit rates being comparatively equal. So for the following experiments, we will compare the PSNR of the two coders and disregard the bit rate.

6.2.1 Results of comparison, without prediction

We will first start by comparing the results from the two RSQ coders when not using any prediction, i.e. the only processing done on each frame is quantization and entropy coding. For this experiment, all pixels will have a value range of $[0, 255]$ (eight bits). To ensure that the output value range is of the same size for the two RSQ coders, we use three bits for the end-zone RSQ coder, which equals 2^3 or eight representation levels. To ensure the same amount of representation levels in the initial RSQ coder, we will use a QP of 32 and use the floor function to limit the output value range to $\left[0, \left\lfloor \frac{255}{32} \right\rfloor\right]$ which equals the range $[0, 7]$, i.e. eight representation levels. Table 6.1 shows the result of the comparison for all three example videos. We see that the end-zone RSQ coder outperforms the initial RSQ coder for all types of videos. The end-zone RSQ coder performs significantly better when coding the videos containing artificial and compound content, which support our previous assumptions.

Video	Initial RSQ	End-zone RSQ
foreman_cif	22.4293	25.7560
screen_content_1	24.7093	26.9117
screen_content_2	22.1369	28.6530

Table 6.1: Comparison of PSNR between end-zone RSQ coder and initial RSQ coder, no prediction.

6.2.2 Results of comparison, with prediction

We will now re-introduce prediction to see how the two coders handle a doubling of the possible range of input values (from potentially $[0, 255]$ to $[-255, 255]$). Both the pixel-wise intra-prediction discussed in Section 4.1 and the per-frame *differential coding* used in the traditional coder was attempted, and the best results was achieved when using differential coding. We will therefore continue using differential coding for the extended end-zone RSQ coder for the following experiments.

Introducing prediction causes problems for the entropy coding when using the end-zone RSQ coder. As the representation levels are divided evenly among the the range $[-255, 255]$, we cannot simply add the minimum value as we have done in our previous coders, as this will cause the maximum input value to be $2 \cdot Max$, which as mentioned is too large for the entropy coder we have used (8-bit maximum value). We thus modify the code to split the signal into a positive part and a negative part by extracting all positive indices and all negative indices from each frame with the built-in function *find()*. We then create two temporary frames that contain each of the types of values. The operation is shown in the code snippet below, where the *codedYFrames* matrix contains the quantized frames:

```
1 plusIndices = find(codedYFrames(:,:,i)>=0);
2 minusIndices = find(codedYFrames(:,:,i)<0);
3 tempFrame = codedYFrames(:,:,i);
4 plusValues = tempFrame(plusIndices);
5 minusValues = tempFrame(minusIndices);
```

The negative values are coded after adding the absolute value of the minimum value in the frame, to ensure that the range is within $[0, 255]$. The positive values are coded directly.

Figure 6.4, Figure 6.6 and Figure 6.5 show the comparison between the two RSQ coders for all three of our sample videos. The initial RSQ coder uses a QP of 32, with the value range of the predicted image being $[-255, 255]$.

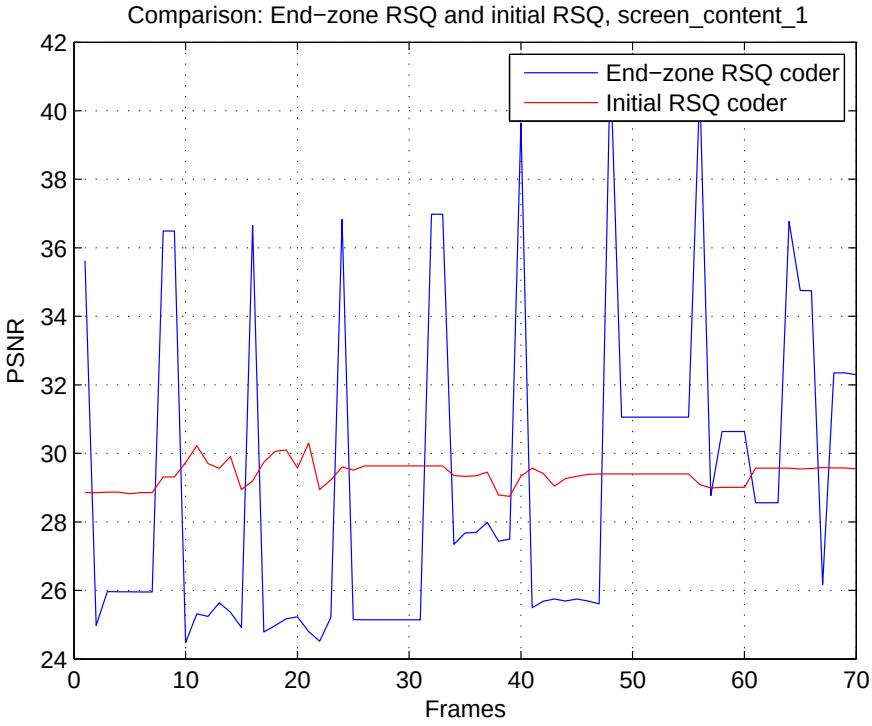

Figure 6.4: A comparison of the initial RSQ coder and the end-zone RSQ coder when coding the 'screen_content_1' video.

This results in a quantized value range of $[-8, 8]$, which is equal to 17 representation values. We will thus pass four bits to the end-zone RSQ coder, which equals 2^4, i.e. 16 representation values, which is close enough for comparison. The I-frame frequency, i.e. the amount of frames between each I-frame, is set to eight.

We see that the initial RSQ-coder is more stable and performs on average well for each frame. The end-zone RSQ coder however, excels at some frames and does a poor job on others, as we see from the spikes in the PSNR. This is especially pronounced in Figure 6.4, where the PSNR is larger than 36 dB for all I-frames, but drops down to $\approx 24.5dB$ for the intermediate (P) frames. We see that this happens for the frames which contain motion, i.e. shifting of the window. The end-zone quantizer does not perform well on predicted frames, but it does an excellent job on non-differentiated (I) frames. The reason for this kind of behavior is that the property of extended end-zones does not work as intended when the values are predicted, i.e. differentiated from an anchor value. If we look at the mean PSNR values in Table 6.2 we

see that the initial RSQ coder actually performs slightly better on average than the extended end-zone quantizer in terms of average PSNR.

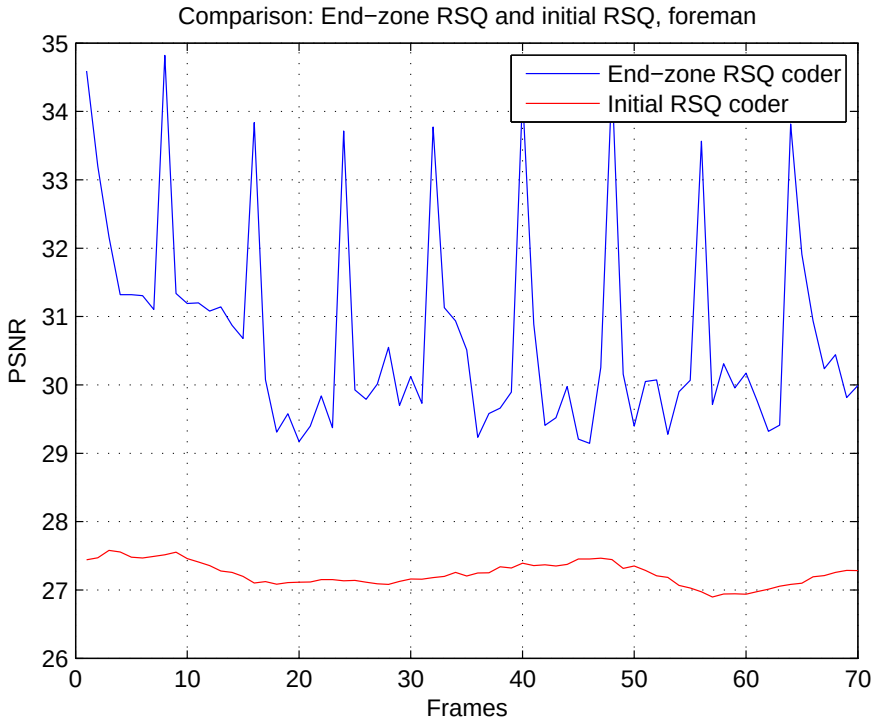

Figure 6.5: A comparison of the initial RSQ coder and the end-zone RSQ coder when coding the 'foreman' video.

Figure 6.5 shows that the end-zone RSQ coder does a better job for all frames when coding natural video, even when coding P frames. The peaks in PSNR still occur at each I-frame, because the dynamic range of P frames are potentially $[-255, 255]$, but for I frames it is only $[0, 255]$, which results in twice the amount of representation levels covering the dynamic range of the values for the equivalent amount of bits, making the comparison somewhat unfair in terms of the initial RSQ coder.

Figure 6.6 shows that the extended end-zone RSQ coder outperforms the initial RSQ coder for most frames. As mentioned in Section 3.2, the content of the 'screen_content_2' video is largely still images with transitions between the web-pages at frames 16 and 38, and the maximizing of the window portraying a PDF file at frame 71. We see that the there is a large drop in PSNR for the end-zone RSQ coder during the web-page transition at frame 38. The transition at frame 16 is largely unaffected, the reason being

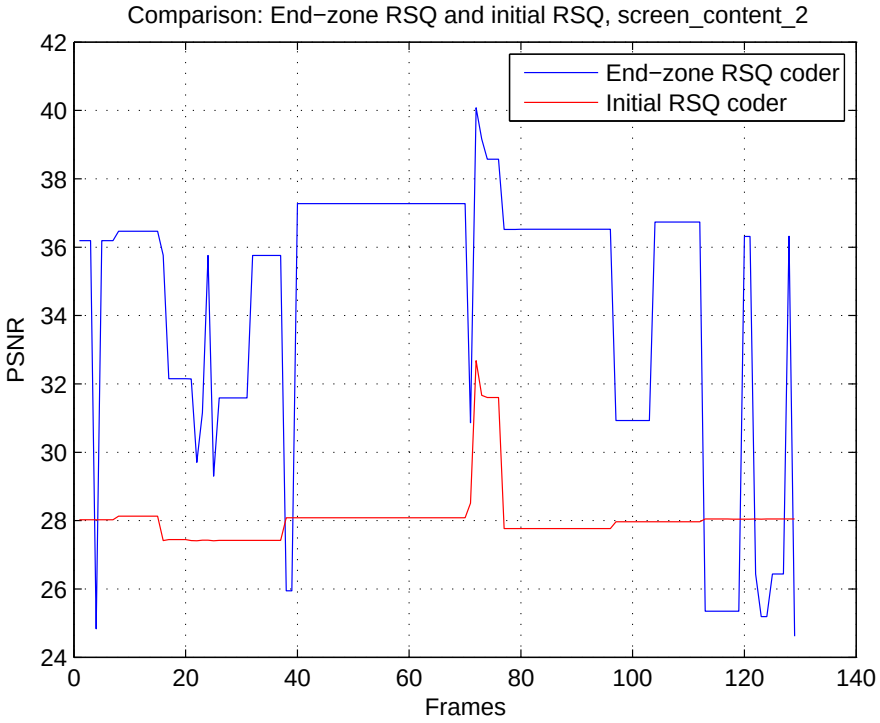

Figure 6.6: A comparison of the initial RSQ coder and the end-zone RSQ coder when coding the 'screen_content_2' video.

that the I-frame frequency is set to eight, and any differential coding during the transition is thus avoided.

Table 6.2 shows the mean PSNR values of each coder. As mentioned we see that the initial RSQ coder slightly outperforms the end-zone RSQ coder when coding the 'screen_content_1 video'. For natural video ('foreman') and compound video ('screen_content_2') the end-zone RSQ coder outperforms the initial RSQ coder significantly.

The end-zone RSQ coders used for comparisons can be found attached under the names 'rsq_coder_ee_no_pred.m' (RSQ coder without prediction) and 'rsq_dpcm_ee.m' (RSQ coder with prediction). The quantizer used can be found under the name 'quantize_rsq.m'. We select between the uniform and extended end-zone quantizer by commenting out the code for the technique we do not wish to use as shown in the code snippet below.

Video	Initial RSQ	End-zone RSQ
foreman_cif	27.2386	30.7364
screen_content_1	29.3961	29.0392
screen_content_2	28.0510	34.4036

Table 6.2: Comparison of PSNR between end-zone RSQ coder and initial RSQ coder, with prediction.

```
%Find representation levels
y=zeros(1,Nlevel);

%% Leave this cell uncommented for regular uniform quantization
% for i = 1:Nlevel
%     y(i)=Min + delta*(2*i-1)/2;
% end

%% Leave this cell uncommented for extended end-zone quantizer

% Set representations levels based on delta
for i = 2:Nlevel-1
    y(i)=Min + delta*(2*i-1)/2;
end
% The first and last representation levels are set to
% the signals minimum and maximum value respectively.
y(1) = Min;
y(Nlevel) = Max;
```

6.3 Problems with end-zone RSQ

As can be seen from the experiments done in the previous section, the end-zone RSQ coder performs sub-optimal when coding differentially predicted frames, i.e. frames where the range of values is increased to include negative values of up to the same magnitude as positive values.

Even if the end-zone quantizer does not perform optimally for differentially coded frames, we see from Figure 6.1 that there is a potential gain to be obtained from tailoring the quantization representation values to better suit the characteristics of the source signal. These characteristics might be known or assumed beforehand, such as we did in Section 6.1 when we assumed that the pixel values in each frame was biased towards 0 and 255. We might also attempt to find these characteristics dynamically for each block or frame.

This will increase the complexity of the coder, but it will also increase the versatility, by not requiring any *a priori* knowledge of the source that is to be coded. We will touch upon this topic again in Chapter 7.

6.4 Implementing end-zone RSQ in the adaptive coder

In this section we will attempt to implement the extended end-zone RSQ coder into the adaptive coder to see if this improves the coding results achieved. We will use a QP of 32 for the traditional coder, and equivalently four bits for the extended end-zone quantizer in the RSQ coder to get the same amount of representation values to cover the range of possible sample values (i.e. $[-255, 255]$). We used the video 'screen_content_2' as source video, an I frame frequency of six ($iFrameFreq = 6$) and a block size of four.

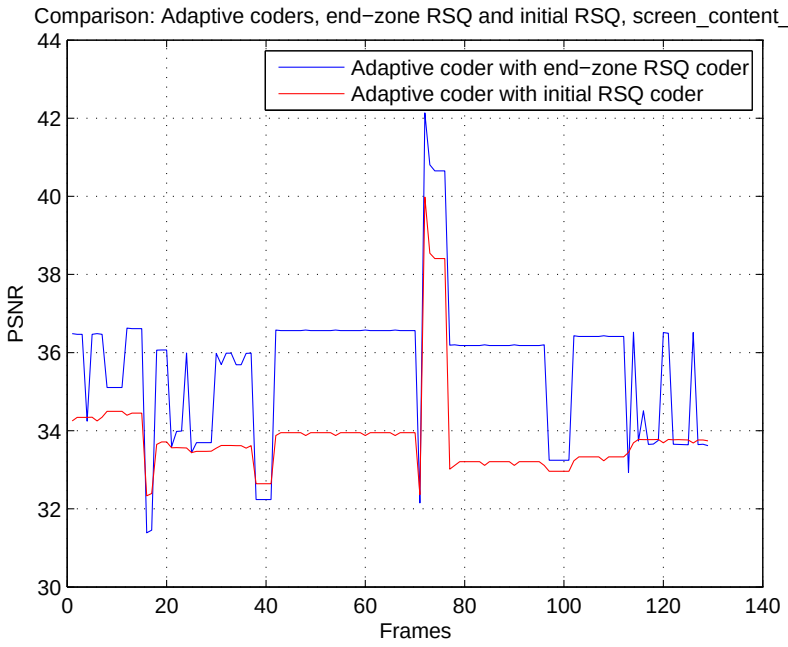

Figure 6.7: Comparison of the two adaptive coders.

Table 6.3 shows that the adaptive quantizer using the end-zone RSQ coder outperforms the initial adaptive coder which was implemented in Chapter 5, both in terms of the mean PSNR and total amount of bits used. Figure

	Initial RSQ	**End-zone RSQ**
Mean PSNR	33.8083	35.6933
Bit rate	10905927	10542456
Lagrange number	18620000	15019000

Table 6.3: Comparison between the adaptive coder using initial RSQ and the adaptive coder using end-zone RSQ. $\lambda = 0.85$.

6.7 shows us the PSNR for each frame of the adaptive coder. The adaptive coder using end-zone RSQ performs on a frame-wise basis better on nearly all frames. Again, the drops in PSNR are caused by the poor performance of the end-zone RSQ coder when coding differentially predicted frames. An even better way of implementing would be to adaptively choose between the initial and end-zone RSQ coders we have introduced in the thesis, such that it chooses the best performing coder for each block (i.e. most likely the end-zone RSQ coder for I-frames and the initial RSQ coder for P frames).

Chapter 7

Future work

This chapter will introduce ideas and improvements that we did not have time to further investigate during our work on this thesis.

7.1 Further optimizing quantization

In this section we will explore some methods to improve the RSQ coder in the presence of difficulties such as the ones mentioned in Section 4.4. In Section 6.1 we proposed an improved quantization method that took the characteristics of the source signal into consideration. However, as discussed in Section 6.2, this quantization method works sub-optimally in terms of PSNR if the signal contains values in the range of $[0, 255]$. A possible solution could be to use a quantization table that is derived from a pre-set QP (similar to H.264), but adapted to better fit each individual block. This may be done by tailoring the quantization steps based on the distribution (e.g. a histogram) and variance of the input signal. Adaptive quantization may introduce gain in visual quality for the reconstructed image, but the cost will be increased complexity as the variance and coefficient value distribution needs to be calculated for each block or frame, increasing the time .

7.2 Base color index map

Another possible improvement when coding artificial content is a technique referred to as *base colors and index map* (BCIM). This technique is more thoroughly discussed in [17] and [9]. In essence, the idea behind the technique is to extract a table of base colors from each block in an image. The base colors are then index mapped to each sample, a sort of color quantization, This is effective because the number of colors in text or graphic blocks are

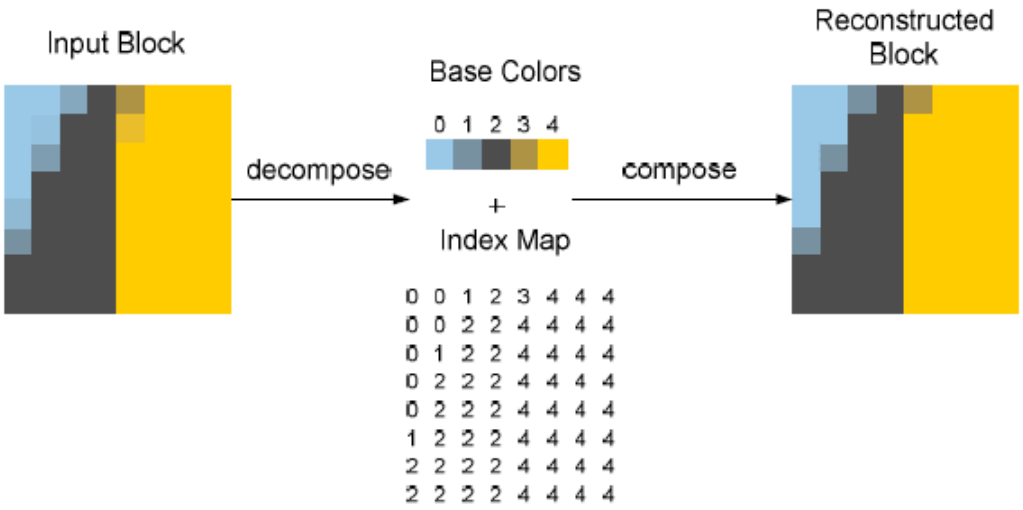

Figure 7.1: Base color table and index mapping (BCIM), taken from [9].

normally limited, resulting in the need of few bits to represent the possible colors in a block. All pixels within an object in CGC will often contain the exact same color, whereas for natural video this is not the case as natural objects are rarely completely uniform in terms of color.

The process of BCIM is shown in Figure 7.1. As we see, we need only a few colors to represent the entire 8x8 block, and the differences between the original block and the reconstructed block are subtle.

7.3 SSIM in RDO

As discussed in Section 4.3 when coding artificial content using the RSQ coder, the coder achieves a lower PSNR even though the coded content does not contain any of the noise artifacts present when coding using the traditional coder when similar parameters are used. This presents a problem when using RDO to select the best possible block. If the mean square error is not necessarily consistent with human perception, how may we be sure that the best possible block is chosen?

The Structured Similarity (SSIM) index is a visual quality metric designed to be more consistent with human perception[21], compared to the MSE which PSNR is based on, as shown in Equation 2.4. It would therefore be an interesting experiment to use SSIM as a distortion metric instead of the

square error when adaptively coding using RDO. Again, this would increase the complexity of the coder, but it may improve the quality of the final reconstructed image as the blocks that are of perceptually higher quality is more likely to be selected. SSIM may also be preferable to PSNR when performing comparison of reconstructed images or videos.

7.4 Deblocking filter on CGC

Experiments were attempted on the use of the deblocking filter when coding CGC, as discussed in Section 2.2.7. We were unfortunately unable to achieve any solid results as the deblocking filter options in the JM reference software lack documentation.

7.5 Implementing proposed changes to H.264

All of the proposed methods introduced in this thesis have only been tested in simplified environments. Therefore no final conclusions may be drawn on how the proposed methods will perform when implemented in a full-scale H.264 environment. The simplest procedure for full-scale testing would be to implement the proposed changes in the JM reference software and then perform the tests.

Chapter 8

Conclusion

We will make our conclusions based on the goals we set in Section 1.3. Each objective will be discussed and analyzed, and finally we will conclude if we have achieved our main goal, which is to improve the visual experience when coding CGC using the H.264 algorithm.

> "Analyze the characteristics of natural video and CGC, and learn how they differ from one another."

We have learned that CGC and natural video exhibit significant differences, and the approach when compressing content of either kind must therefore be tailored specifically for the content to be coded. The nature of natural video is usually inherently isotropic with smooth transitions, thus the content consists of mostly low-frequency components. Computer generated content is regarded as anisotropic, with sharp and regular edges, resulting in content consisting of higher frequency components than natural video.

> "Learn how the H.264 coder actually performs when coding CGC at low bit rates."

The JM reference software, which is an implementation of the H.264 standard, was used to code both a sequence of natural video and a sequence of artificial content. The resulting compressed natural video required less than half the bit rate and achieved a better visual quality (measured in PSNR) compared to the coded artificial content. The sequences were of the same length and resolution and the same quantization parameter were used for both coding processes. We thus concluded that the H.264 coding algorithm indeed performed better for natural video than artificial content for our sample content.

> "Classify the noise and artifacts that occurs when coding CGC using the regular H.264 algorithm."

We learned that quantizing in the transform domain is analogous to truncating the series expansion of the component transform, thus reducing the accuracy of the transform. This in turn leads to increased oscillations around the points of discontinuity, commonly known as Gibbs phenomenon. This phenomenon is visible as ringing artifacts around sharp edges and high frequency components, such as text and graphical objects. Because the H.264 obtains compression through the removal/reduction of high frequency components, we also get a slight blurring of edges and objects. The avoidance of these noisy components is thus the main problem that must be solved for our goal to be met.

> "Identify which components in the H.264 algorithm that causes the sub-optimal performance when coding CGC."

H.264 uses a component transform to code natural content as the information contained in the transformed content gets packed together allowing for more efficient compression. However, this only works because of the isotropic nature of natural content. Because the nature of CGC is inherently anisotropic, performing a component transform leads to the signal becoming less compact. We have therefore identified the component transform as the main cause of the sub-optimal performance in the H.264 algorithm when coding CGC, and we thus propose to omit the component transform and quantize the residual signal directly.

> "Investigate how these components might be modified and improved to better take into consideration the characteristics of the CGC."

By slightly modifying the intra prediction scheme and quantizing the residual signal directly, we were able to achieve better coding results than when using the traditional "transform first, quantize later" scheme of the standard H.264 algorithm for some CGC. However, even when the traditional coding scheme performed better in terms of PSNR, none of the artifacts and noise present when coding with the traditional scheme occurred when coding using the proposed method. A new quantization technique was also proposed, denoted *extended end-zone quantization*. This technique was tailored to better match the characteristics of the source signal that was being quantized. Experiments showed that the extended end-zone quantization obtained better results than the regular uniform quantizer.

> "Introduce a method that determines if the source that is to be coded consists of natural video, CGC or a mix of the two. Then code the content accordingly."

We introduced an adaptive scheme that coded each block in a frame using both the traditional idea of first performing a component transform on the signal followed by quantization, and the proposed idea of quantizing the residual signal directly. We then perform Lagrange optimization on each block and the block achieving the minimum Lagrange value was selected to be used in the reconstructed frame. The results from the experiments conducted using this scheme showed that it outperformed each of the stand-alone coders.

Looking at each objective and the results we have achieved, we may finally conclude that the methods proposed in this thesis may indeed improve the performance of the H.264 algorithm when coding CGC. However, we cannot make any definite conclusions until we implement the proposed techniques in a full-scale H.264 implementation, to take all aspects of the H.264 algorithm into consideration. Unfortunately, this proved to be too complex to accomplish in the allotted time slot and will thus be left as a topic for future research.

Appendix A

Tools

This section contains a list of the tools used in this thesis. A further explanation of their areas of use is given when they are first introduced in the thesis.

- **Software:**
  - MATLAB 7.11.0 - high-level language and interactive environment. Used for all programming. Downloaded from [24].
  - JM reference software 17.2 - open source H.264 reference software. Used to encode/decode .yuv files using the H.264 standard. Downloaded from [37].
  - CamStudio 2.0 - used to record content on the computer screen. Downloaded from [7].
  - YUVTools 3.0 - used to convert between .avi and .yuv, in addition to playing .yuv files. It can also be used to compare PSNR between two video files. Downloaded from [40].
  - ffmpeg - used to convert between .264 and .yuv. Downloaded from [1].
- **MATLAB libraries**
  - YUV Toolbox - For converting RGB frames to YUV frames, in addition to exporting video to .yuv. Downloaded from [39].
  - Huffman - Used to convert blocks or frames to Huffman codewords. May also be used to create lists of bit codes. Downloaded from [34].

Appendix B

JM Reference Software Parameters

This appendix contains a list of the JM reference software parameters that have been used in this thesis.

- FramesToBeEncoded - numbers of frames to be coded
- SourceWidth - Source frame width (352 for CIF)
- SourceHeight - Source frame height (288 for CIF)
- ProfileIDC - Profile IDC (66=baseline, 77=main, 88=extended; FREXT Profiles: 100=High, 110=High 10, 122=High 4:2:2, 244=High 4:4:4, 44=CAVLC 4:4:4 Intra, 118=Multiview High Profile, 128=Stereo High Profile)

  (This must be set to match the desired color downsampling format, in our case 100 (4:2:0) or 122 (4:2:2))
- IntraPeriod - Period of I-pictures (To skip motion compensation, this can be set as 1)
- QPISlice - Quant. param for I Slices (This is a value between 0-51, a higher number means coarser quantization resulting in a lower bitrate but reduced visual quality)
- QPPSlice - Quant. param for P Slices
- QPBSlice - Quant. param for B Slices
- InitialQP - Initial Quantization Parameter for the first I frame

- YUVFormat - YUV format (0=4:0:0, 1=4:2:0, 2=4:2:2, 3=4:4:4)
- Interleaved - 0: Planar input, 1: Packed input
- NumberBFrames - Number of B coded frames inserted in a row between I-frames and P-frames (0=not used)

Appendix C

List of attached files

This appendix contains a list of some of the material that was used in this thesis, a long with a description of each file.

- **Videos:**
  - *screen_content_1* - Experiment video, consists of purely screen content.
  - *screen_content_2* - Experiment video, screen content with embedded natural images.
  - *foreman* - Experiment video, natural video.
  - *traditional_scc2* - *'screen_content_2'* coded using the traditional coder. Used in Section 4.3.
  - *rsq_scc2* - *'screen_content_2'* coded using RSQ coder. Used in Section 4.3.
- **Coders:**
  - *'trad_coder_no_pred.m'* - Traditional coder (2D DCT and quantization), no prediction used. Used in Section 4.3.
  - *'rsq_coder_no_pred.m'* - RSQ coder, no prediction used. Used in Section 4.3.
  - *'rsq_coder_ee_no_pred.m'* - RSQ coder, extended end-zone quantizer, no prediction used. Used in Section 6.2.
  - *'rsq_dpcm_ee.m'* - RSQ coder, extended end-zone quantizer, framewise differential prediction used. Used in Section 6.2.2.
  - *'adpt_coder_intra_pred_rsq.m'* - The adaptive coder used in Section 5.2.

  - *'adpt_coder_ee_rsq.m'* - Adaptive coder using the extended end-zone RSQ coder, used in Section 6.4.
  - *'quantizer.m'* - Performs 2D DCT and uniform quantization on a block.
  - *'iQuantizer.m'* - Performs inverse quantization and inverse 2D DCT on a block.
  - *'quantize_rsq.m'* - Contains the uniform and extended end-zone quantizer.
  - *'quantizerNoDCT.m'* - The quantizer for the initial RSQ coder.
  - *'iQuantizerNoDCT.m'* - The inverse quantizer for the initial RSQ coder.
  - *'psnrcalc.m'* - Calculates the PSNR of the reconstructed frame.

- **Images:** The images listed here are all used in the thesis, however, because of constraints on the size, they are also attached so that they made be viewed separately in case the details of the image are difficult to distinguish.

  - *'adaptive_trad_comparison.jpg'* - A still image from the sequence *'screen_content_1'* coded with the traditional coder using a QP of 25. The image was used in Section 5.2.
  - *'adaptive_rsq_comparison.jpg'* - A still image from the sequence *'screen_content_1'* coded with the RSQ coder using a QP of 25. The image was used in Section 5.2.
  - *'adaptive_comparison.jpg'* - A still image composed of the optimal blocks in terms of visual quality and measured bits from the 'adaptive_trad_comparison.jpg' and 'adaptive_rsq_comparison.jpg' images. The image was used in Section 5.2.
  - *'comp_cg_qp_40_mvc.png'* - A still image coded using the JM reference software. The source signal used was *'screen_content_1'*. The image was used in Section 3.4. This image is also attached in this appendix.
  - *'rsq_vs_trad_no_pred.png'* - A comparison between the traditional coder and the RSQ coder with no use of prediction. The frame is from the sequence *'screen_content_1'*. The image was used in Section 4.3. This image is also attached in this appendix.

- **Websites** A .zip file containing all web sites that have referenced is also attached, such that all sources that have been cited may be controlled.

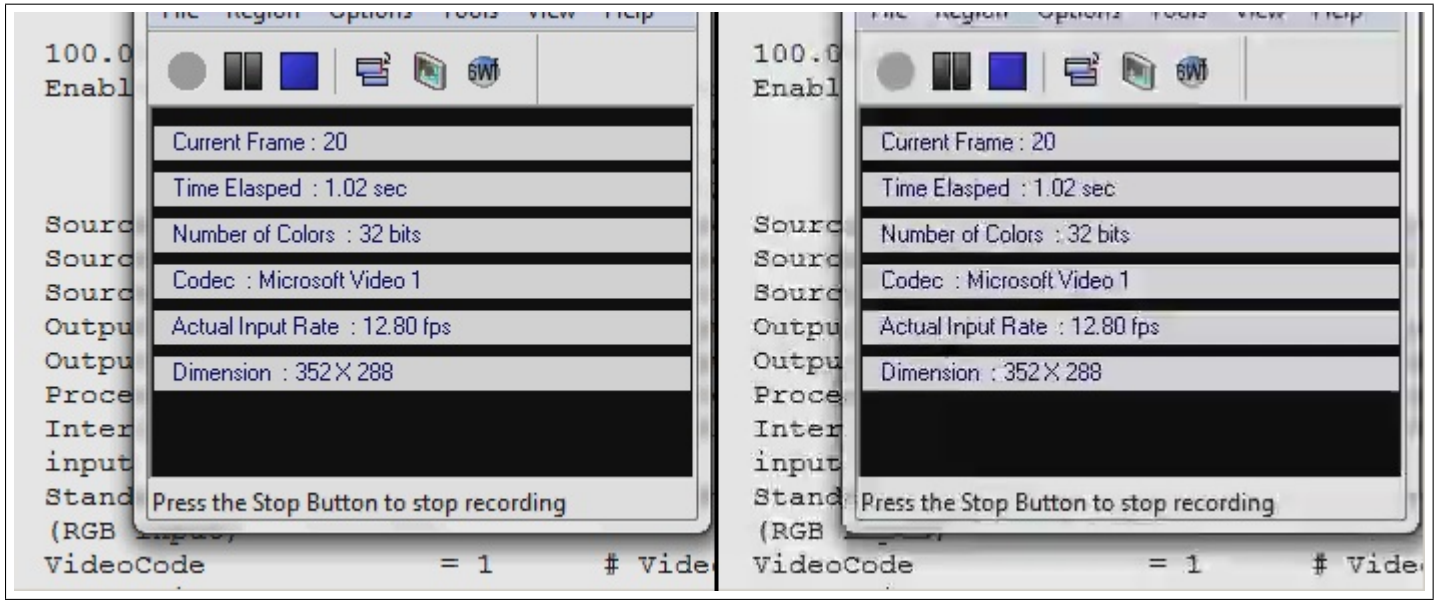

Figure C.1: Enlarged version of Figure 3.5 used in Section 3.3.

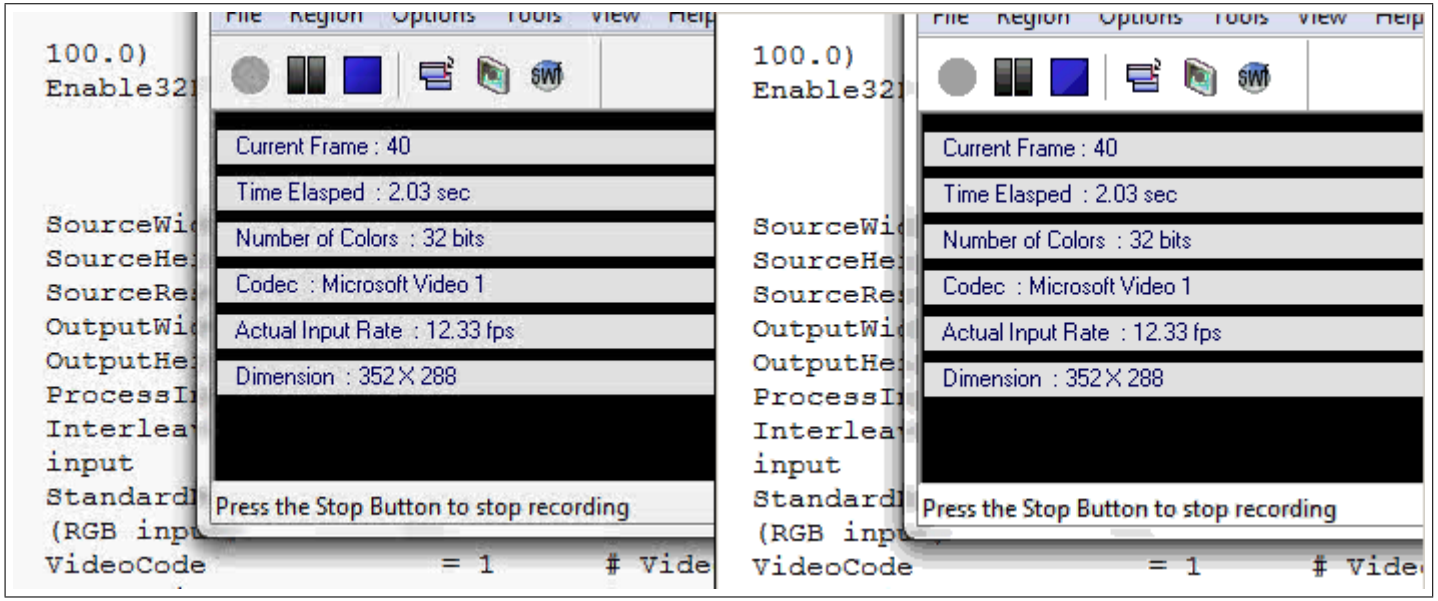

Figure C.2: Enlarged version of Figure 4.2 used in Section 4.3.

Appendix D

MATLAB Code

This chapter contains the MATLAB code for one of the coders created. The coder chosen as an example is the adaptive coder using extended end-zone RSQ coding. The reason this coder was chosen is that it is the most complex and it achieved the best results, and therefore is probably the most interesting.

```
%% Adaptive coder
clc
clear all

% Read avi file
% Available movies are
% 'screen_content_1.avi', 352x288 resolution, 73 frames
% 'screen_content_2.avi', 720x480 resolution, 129 frames
% 'foreman_cif.avi', 352x288 resolution, 300 frames
movObj = mmreader('screen_content_1.avi');

% Convert to 4D matrix with coefficients
vidFrames = read(movObj);

display('Adaptive coding')

% Find height, width and number of frames
sizeOfMatrix = size(vidFrames);
nHeight = sizeOfMatrix(1);
nWidth = sizeOfMatrix(2);
nFrames = sizeOfMatrix(4);

% Set I-frame frequency (GOP size) and lambda (Lagrange
% multiplier) in addition to the quantization parameter
% (totalq) and amount of bits sent to the extended
% end-zone quantizer.
totalq = 32;
```

```
28 quant_factor = 4;
29 iFrameFreq = 6;
30 lambda = 0.85;
31
32 % Size of blocks to be coded
33 bSize = 4;
34
35 % Declare variables that can contain information about frames
36 rFr = uint8(zeros(nHeight, nWidth,nFrames));
37 gFr = uint8(zeros(nHeight, nWidth,nFrames));
38 bFr = uint8(zeros(nHeight, nWidth,nFrames));
39
40 yFrames = zeros(nHeight, nWidth,nFrames);
41 uFr = uint8(zeros(nHeight, nWidth,nFrames));
42 vFr = uint8(zeros(nHeight, nWidth,nFrames));
43
44 yCells = cell(1,nFrames);
45 rsqYCells = cell(1,nFrames);
46 tradYClls = cell(1,nFrames);
47 uCells = cell(1,nFrames);
48 vCells = cell(1,nFrames);
49
50 % Split video into R, G and B frames
51 for i = 1:nFrames
52   rFr(:,:,i) = vidFrames(:,:,1,i);
53   gFr(:,:,i) = vidFrames(:,:,2,i);
54   bFr(:,:,i) = vidFrames(:,:,3,i);
55   tmpR = rFr(:,:,i);
56   tmpG = gFr(:,:,i);
57   tmpB = bFr(:,:,i);
58   % Convert from RGB to YUV
59   [yFrames(:,:,i),uFr(:,:,i),vFr(:,:,i)]=rgb2yuv(tmpR,tmpG,tmpB);
60 end
61
62 % Declare variables to pre-allocate memory
63 blocks = zeros(bSize,bSize,nWidth*nHeight/(bSize.^2));
64 rcTrdYFrs = zeros(size(yFrames));
65 rcRSQYFrs = zeros(size(yFrames));
66 rcYFrs = zeros(size(yFrames));
67 dpcmBlocks = zeros(size(blocks));
68 quantTradBlocks = zeros(size(blocks));
69 iQuantTradBlocks = zeros(size(blocks));
70 rcTrdBlcks = zeros(size(blocks));
71 tempBlock = zeros(bSize,bSize);
72 tempBlockTrad = zeros(bSize,bSize);
73 rdTrad = zeros(2,length(blocks));
74 rdRSQ = zeros(2,length(blocks));
75 rcTmpFrm = zeros(bSize,bSize);
76
```

```
77  % These will contain the PSNR values of the reconstructed frames
78  %for each coder.
79  psnrVector = zeros(1,nFrames);
80  rsqPSNRVector = zeros(1,nFrames);
81  tradPSNRVector = zeros(1,nFrames);
82
83  % These  variables contain the total bits used for each coder.
84  totalRateRSQ = 0;
85  totalrateTrad = 0;
86  totalRateAdaptive = 0;
87
88  % Iterate over all frames and encode each frame
89  for i = 1:nFrames
90    l = 1;
91    j = bSize;
92    k = bSize;
93    % Split frames into blocks and create a 3D matrix
94    % to contain all blocks.
95    for p = bSize:bSize:nWidth*nHeight/bSize
96      blocks(:,:,l) = yFrames(k-(bSize-1):k,j-(bSize-1):j,i);
97      k = k + bSize;
98        if mod(p,nHeight) == 0
99          j = j+bSize;
100         k = bSize;
101       end
102       l = l+1;
103   end
104   % Differential coding. Skip differential coding for initial
105   % frame and each I-frame, determined by I-frame frequency
106   % (iFrameFreq).
107   if i == 1 || mod(i,iFrameFreq) == 0
108     dpcmBlocks = blocks;
109     anchorBlocks = blocks;
110   else
111     dpcmBlocks = blocks - anchorBlocks;
112   end
113
114   rateTrad = 0;
115   % Traditional coder, iterate over each block in the frame.
116   for p = 1:length(dpcmBlocks)
117     % 2D DCT transform and uniform quantization done in
118     % quantizer(blocks, qp) function.
119     quantTradBlocks(:,:,p) = quantizer(dpcmBlocks(:,:,p),totalq);
120
121     % Entropy code quantized blocks
122     % Extract minimum value and add it to the values to be
123     % entropy coded, to ensure that no values are less than 0.
124     huffBlock = quantTradBlocks(:,:,p);
125     minVal = abs(min(huffBlock(:)));
```

```
126     [tradCell,info]=norm2huff(uint8(huffBlock(:)+minVal));
127     % We split the huffcodes into indices and huff values, the
128     % indices will not be used.
129     [tmpk,tmpl,huff] = find(info.huffcodes);
130     % We pass the huffcodes to huffcodes2bin to create a list
131     % of bit codes.
132     [listOfBitCodes,symbols] = huffcodes2bin(info.huffcodes);
133     sortedCodes = zeros(1,length(huff));
134     % Sort frequency of codes to match huffman bit codes
135     % (most frequent => shortest bit codes)
136     for j = 1:length(huff)
137       sortedCodes(j) = sum(sum(quantTradBlocks(:,:,p)==huff(j)));
138     end
139     sortedCodes = sort(sortedCodes,'descend');
140     bitsPrBlck = 0;
141     tmpBits = 0;
142     % Multiply the length of bit codes with the frequency of the
143     % value to get the amount of bits used. Add the
144     % bit-representation of the actual value.
145     for j = 1:length(huff)
146       tmpBits = length(listOfBitCodes{j})*sortedCodes(j);
147       bitsPrBlck = bitsPrBlck+tmpBits+ceil(log2(huff(j)));
148     end
149     if sum(tradCell(:)) == 0
150       % If the sum of rsqCell is 0, all values in the Huffman
151       % coded block are equal and the Huffman coder does not
152       % work properly, consequently we have to add the value
153       % (minVal) of the block that was coded to the inversely
154       % entropy coded block.
155       rcTmpFrm(:)=minVal;
156       tempTradBlock = quantTradBlocks(:,:,p);
157       % If the uniform value is non-zero, add the given
158       % bit-representation of the value to the amount
159       % of bits used. Otherwise add 1 bit.
160       if var(tempTradBlock(:)) ~= 0
161         bitsPrBlck=bitsPrBlck+ceil(log2(minVal));
162       else
163         bitsPrBlck = bitsPrBlck + 1;
164       end
165     else
166       % Inverse entropy coding and add the minVal which was
167       % subtracted before the entropy coding was done.
168       rcTmpFrm(:) = double(huff2norm(tradCell,info))-minVal;
169     end
170     rateTrad = rateTrad + bitsPrBlck;
171     % Inverse quantization
172     iQuantTradBlocks(:,:,p) = iQuantizer(rcTmpFrm,totalq);
173     % Store the rate for each block
174     rdTrad(1,p) = bitsPrBlck;
```

```
175   end
176
177   rateRSQ = 0;
178   quantizedRSQBlocks = zeros(size(blocks));
179   iQuantRSQBlocks = zeros(size(blocks));
180   % Perform quantization on entire frame unless it consist of
181   % all 0 coefficients, which will cause the coder to break down.
182   if var(dpcmBlocks(:) ~= 0)
183     tempVector = quantize_rsq(dpcmBlocks(:), quant_factor);
184     quantizedRSQBlocks(:) = tempVector(:);
185   else
186     quantizedRSQBlocks(:) = 0;
187   end
188
189   % RSQ coding
190   for p = 1:length(quantizedRSQBlocks)
191     % Split block into plus and minus values, which are coded
192     % seperately and independently.
193     plusIndices = find(quantizedRSQBlocks(:,:,p)>=0);
194     minIndices = find(quantizedRSQBlocks(:,:,p)<0);
195     tempFrame = quantizedRSQBlocks(:,:,p);
196     minVal = abs(min(tempFrame(:)));
197     plusValues = tempFrame(plusIndices);
198     minVals = tempFrame(minIndices);
199     bitsPrBlck = 0;
200     % Entropy code positive quantized values (if any)
201     if ~isempty(plusValues)
202       [rsqCell, info] = norm2huff(uint8(plusValues(:)));
203       [tmpk,tmpl,huff] = find(info.huffcodes);
204       [listOfBitCodes,symbols] = huffcodes2bin(info.huffcodes);
205       sortedCodes = zeros(1,length(huff));
206       % Sort frequency of codes to match huffman bit codes
207       % (most frequent => shortest bit codes)
208       for j = 1:length(huff)
209         sortedCodes(j) = sum(sum(plusValues(:,:)==huff(j)));
210       end
211       sortedCodes = sort(sortedCodes,'descend');
212       tmpBits = 0;
213       % Multiply the length of bit codes with the frequency of
214       % the value to get the amount of bits used. Add the
215       % bit-representation of the actual value.
216       for j = 1:length(huff)
217         tmpBits = length(listOfBitCodes{j})*sortedCodes(j);
218         bitsPrBlck=bitsPrBlck+tmpBits+ceil(log2(huff(j)));
219       end
220     end
221     % Entropy code negative quantized values (if any)
222     if ~isempty(minVals)
223       tempMinus = minVals + minVal;
```

```
224         [minusCell, minusInfo] = norm2huff(uint8(tempMinus(:)));
225         [k2,l2,minHuff] = find(minusInfo.huffcodes);
226         minListOfBitCodes = huffcodes2bin(minusInfo.huffcodes);
227         minSortCodes = zeros(1,length(minHuff));
228         % Sort frequency of codes to match huffman bit codes
229         %(most frequent => shortest bit codes)
230         for j = 1:length(minHuff)
231           minSortCodes(j) = sum(sum(tempMinus==minHuff(j)));
232         end
233         minSortCodes = sort(minSortCodes,'descend');
234         tmpBits = 0;
235         % Multiply the length of bit codes with the frequency of
236         % the value to get the amount of bits used. Add the
237         % bit-representation of the actual value.
238         for j = 1:length(minHuff)
239           tmpBits=length(minListOfBitCodes{j})*minSortCodes(j);
240           bitsPrBlck=bitsPrBlck+tmpBits+ceil(log2(minHuff(j)));
241         end
242       end
243       tempBlock = zeros(bSize,bSize);
244       rcTmpFrm = zeros(bSize,bSize);
245       % Inversely entropy code eventual positive values.
246       if ~isempty(plusIndices)
247         % If the sum of rsqCell is 0, all values in the Huffman
248         % coded block are equal and the Huffman coder does not
249         % work properly, consequently we have to add the value
250         % (minVal) of the block that was coded.
251         if sum(rsqCell(:)) == 0
252           % Entire block consists of minVal
253           tempBlock(:) = double(minVal);
254           % Add one bit for each 0-block for comparison.
255           bitsPrBlck =  bitsPrBlck + 1;
256         else
257           % Create a block (tmp) which holds the inversely
258           % entropy coded values.
259           tmp = double(huff2norm(uint8(rsqCell(:)),info));
260           tempBlock(1:length(tmp)) = double(tmp);
261         end
262         % Insert the positive values in their correct positions.
263         rcTmpFrm(plusIndices)=tempBlock(1:length(plusIndices));
264       end
265       % Inversely entropy code eventual negative values.
266       if ~isempty(minIndices)
267         minVals = double(huff2norm(uint8(minusCell(:)),minusInfo));
268         % Insert negative values in their correct positions.
269         rcTmpFrm(minIndices)=minVals(1:length(minIndices))-minVal;
270       end
271       if bitsPrBlck == 0 % Block is all zeros
272         bitsPrBlck = bitsPrBlck + 1;
```

```
273       end
274       rateRSQ = rateRSQ + bitsPrBlck;
275       iQuantRSQBlocks(:,:,p) = rcTmpFrm;
276       rdRSQ(1,p) = bitsPrBlck;
277     end
278
279     % Inverse differential coding
280     if i == 1 || mod(i,iFrameFreq) == 0
281       rcTrdBlcks = iQuantTradBlocks;
282       recAnchorBlocks = iQuantTradBlocks;
283       rcRSQBlcks = iQuantRSQBlocks;
284       rsqAnchorBlocks = iQuantRSQBlocks;
285     else
286       rcTrdBlcks = iQuantTradBlocks + recAnchorBlocks;
287       rcRSQBlcks = iQuantRSQBlocks + rsqAnchorBlocks;
288     end
289
290
291     % Reconstruct frame
292     j = bSize;
293     k = bSize;
294     adaptiveRate = 0;
295     for p = 1:length(blocks)
296       % Calculate distortion for RDO, traditional coder
297       dist=sum(sum(rcTrdBlcks(:,:,p)-blocks(:,:,p))).^2;
298       rdTrad(2,p) = dist;
299       % Calculate distortion for RDO, RSQ coder
300       dist=sum(sum(rcRSQBlcks(:,:,p) - blocks(:,:,p))).^2;
301       rdRSQ(2,p) = dist;
302       % Calculate Lagrange values
303       lagrangeRSQ = rdRSQ(2,p)+lambda.*rdRSQ(1,p);
304       lagrangeTrad = rdTrad(2,p) + lambda.*rdTrad(1,p);
305       % Insert the block with the lowest Lagrange value
306       if lagrangeRSQ < lagrangeTrad
307         rcYFrs(k-(bSize-1):k,j-(bSize-1):j,i)=rcRSQBlcks(:,:,p);
308         adaptiveRate = adaptiveRate + rdRSQ(1,p);
309       else
310         rcYFrs(k-(bSize-1):k,j-(bSize-1):j,i)=rcTrdBlcks(:,:,p);
311         adaptiveRate = adaptiveRate + rdTrad(1,p);
312       end
313       % Create RSQ and traditional coder frames for comparison
314       % purposes.
315       rcTrdYFrs(k-(bSize-1):k,j-(bSize-1):j,i)=rcTrdBlcks(:,:,p);
316       rcRSQYFrs(k-(bSize-1):k,j-(bSize-1):j,i)=rcRSQBlcks(:,:,p);
317       k = k + bSize;
318       if mod(p,nHeight/bSize) == 0
319         j = j+bSize;
320         k = bSize;
321       end
```

```
322   end
323
324   % Print PSNR and bits used for each coder
325   tradPSNR = psnrcalc(yFrames(:,:,i),rcTrdYFrs(:,:,i));
326   tradPSNRVector(i) = tradPSNR;
327   totalrateTrad = totalrateTrad + rateTrad;
328   fprintf('Traditional coder PSNR: %d \n',tradPSNR);
329   fprintf('Traditional coder rate: %d \n',rateTrad);
330
331   rsqPSNR = psnrcalc(yFrames(:,:,i),rcRSQYFrs(:,:,i));
332   rsqPSNRVector(i) = rsqPSNR;
333   totalRateRSQ = totalRateRSQ + rateRSQ;
334   fprintf('RSQ coder PSNR: %d \n',rsqPSNR);
335   fprintf('RSQ coder rate: %d \n',rateRSQ);
336
337   adaptivePSNR = psnrcalc(yFrames(:,:,i),rcYFrs(:,:,i));
338   psnrVector(i) = adaptivePSNR;
339   totalRateAdaptive = totalRateAdaptive + adaptiveRate;
340   fprintf('Adaptive coder PSNR: %d \n',adaptivePSNR);
341   fprintf('Adaptive coder rate: %d \n',adaptiveRate);
342   fprintf('New frame\n');
343 end
344
345 % Create yuv file
346 for j = 1:nFrames
347   % Convert to cells for yuv exporting
348   yCells(j) = mat2cell(rcYFrs(:,:,j),nHeight,nWidth);
349   rsqYCells(j) = mat2cell(rcRSQYFrs(:,:,j),nHeight,nWidth);
350   tradYClls(j) = mat2cell(rcTrdYFrs(:,:,j),nHeight,nWidth);
351   uCells(j) = mat2cell(uFr(:,:,j),nHeight,nWidth);
352   vCells(j) = mat2cell(vFr(:,:,j),nHeight,nWidth);
353 end
354
355 % Export to yuv file
356 yuv_export(yCells,uCells,vCells,'output.yuv',nFrames,'w');
357 yuv_export(rsqYCells,uCells,vCells,'output_rsq.yuv',nFrames,'w');
358 yuv_export(tradYClls,uCells,vCells,'output_trd.yuv',nFrames,'w');
```

The following function is the 2D DCT transform and the uniform quantizer used in the traditional coder:

```
1 function quantizedBlock = quantizer(inputBlock,qp)
2     % Takes in signal to be quantized and quantization
3     % parameter.
4     % Returns quantized signal.
5
6     % Perform 2D DCT on the block
7     transformedBlock = dct2(inputBlock);
8     % Quantize and round to nearest integer
```

```
    quantizedBlock = round(transformedBlock./qp);
end
```

The following function is the inverse quantization and the inverse 2D DCT transform used in the traditional coder:

```
function reconstructedBlock = iQuantizer(inputBlock,qp)
    % Takes in quantized signal and quantization parameter.
    % Returns reconstructed signal.

    % Inverse quantization
    invQBlock = inputBlock.*qp;
    % Inverse transform after
    reconstructedBlock = idct2(invQBlock);
end
```

The 'quantize_rsq.m' function is the uniform quantizer/extended end-zone quantizer used in the RSQ coder:

```
function [signal_q, delta]=quantize_rsq(signal,Nbit)
% Takes in the signal to be quantized and the amount
% of bits used for quantization. Returns the quantized
% signal along with delta (quantization step size).

% Quantize signal with Nbit using uniform quantization
% adapted to the signal dynamic range.
% Operates on 1- and 2-dimensional signals.
% For 2D signals, it quantizes each row separately.

%Find delta
Min=min(min(signal));
Max=max(max(signal));
Nlevel=2^Nbit;
delta=(Max-Min)/Nlevel;

%Find representation levels
y=zeros(1,Nlevel);

%% Uncomment cell for regular uniform quantization
% for i = 1:Nlevel
%     y(i)=Min + delta*(2*i-1)/2;
% end

%% Uncomment cell for extended end-zone quantizer

% Set representations levels based on delta
for i = 2:Nlevel-1
    y(i)=Min + delta*(2*i-1)/2;
end
% The first and last representation levels are set to
```

```
% the signals minimum and maximum value respectively.
y(1) = Min;
y(Nlevel) = Max;


%%
[M,N]=size(signal);
signal_q=zeros(M,N);

% Iterate over signal to map signal values to
% representation values.
for m=1:M
    for n=1:N
        index=ceil((signal(m,n)-Min)/delta);
        if index==0
            index=1;
        end
        signal_q(m,n)=round(y(index));
    end
end
```

The following function calculates the PSNR between an original frame and a reconstructed frame, if the frames are equal the function returns an error message:

```
function psnr = psnrcalc(origFrame,recFrame)
  % This function takes in the original frame and the
  % reconstructed frame and calculates the PSNR based
  % on the distortion in the reconstructed frame.

  % If the frames are equal, the PSNR is infinite
  if origFrame == recFrame
    error('Images are identical: PSNR has infinite value')
  end
  psnr=10*log10(255^2/(mean((origFrame(:)-recFrame(:)).^2)));
end
```

Bibliography

[1] FFmpeg. http://ffmpeg.org/download.html. Retrieved at 17.01.2011.

[2] foreman_cif. http://www.tkn.tu-berlin.de/research/evalvid/cif/foreman_cif.264. Retrieved at 07.03.2011.

[3] hack4fun. http://www.hack4fun.org/h4f/blog, April 2009. Retrieved at 30.05.2011.

[4] Blu-ray Disc Association. *Application Definition Blu-ray Disc Format*, March 2005. Retrieved from http://www.blu-raydisc.com/Assets/Downloadablefile/bdj_gem_application_definition-15496.pdf.

[5] Bojana Gajić. Kvantisering. Lecture notes, TTT4110 Signal Processing and Communication, 2011.

[6] C. A. Bouman. Lab 8 - number representation and quantization. http://cnx.org/content/m18085/latest/, September 2009. Retrieved at 24.05.2011.

[7] CamStudio. http://www.camstudio.org/CamStudio20.exe, 2009. Retrieved at 11.01.2011.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, second edition, 2003.

[9] W. Ding, Y. Lu, and F. Wu. Enable efficient compound image compression in H.264/AVC intra coding. *IEEE International Conference on Image Processing*, 2:337–340, October 2007.

[10] J. Gibson, T. L. T. Berger and, R. Baker, and D. Lindbergh. *Digital Compression for Multimedia: Principles & Standards*. Morgan Kaufmann, first edition, 1998.

[11] D. Grois, E. Kaminsky, and O. Hadar. Efficient real-time video-in-video insertion into a pre-encoded video stream. *ISRN Signal Processing*, pages 1–11, December 2010. vol. 2011, Article ID 975462.

[12] Q. Hyunh-Thu and M. Ghanbari. Scope of validity of PSNR in image/video quality assessment. *Electronics Letters*, 44(13):800–801, June 2008.

[13] M. Ito. `http://codesequoia.wordpress.com/2009/10/18/h-264-stream-structure/`, October 2009. Retrieved at 24.05.2011.

[14] ITU-T. Objective perceptual multimedia video quality measurement in the presence of a full reference. *Recommendation ITU-T J.247*, page 15, August 2010.

[15] K. Jack. *Video Demystified: A Handbook for the Digital Engineer*. Newnes, fifth edition, 2007.

[16] C. Lan, G. Shi, and F. Wu. Compress compound images in H.264/MPEG-4 AVC by exploiting spatial correlation. *IEEE Trans. on Image Processing*, April 2010.

[17] C. Lan, J. Xu, F. Wu, and G. Sullivan. Screen content coding results using TMuC. *JCTVC-C276*, 3rd JCTVC meeting, Guangzhou, CN:7–15, July 2010.

[18] X. Li, P. Amon, A. Hutter, and A. Kaup. Lagrange multiplier selection for rate-distortion optimization in SVC. *Picture Coding Symp*, pages 1–4, May 2009.

[19] H. Lohscheller. A subjectively adapted image communication system. *IEEE Trans. Commun COM-32*, pages 1316–1322, December 1984.

[20] J. Loomis and M. Wasson. VC-1 technical overview. `http://www.microsoft.com/windows/windowsmedia/howto/articles/vc1techoverview.aspx`, January 2007. Retrieved at 07.04.2011.

[21] D. Lv, D. Y. Bi, and Y. Wang. Image quality assessment based on DCT and structural similarity. *Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference*, pages 1–4, September 2010.

[22] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-complexity transform and quantization in H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, July 2003.

[23] D. Marpe, H. Schwarz, and T. Wiegand. Context-Adaptive Binary Coding in the H.264/AVC video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, July 2003.

[24] Mathworks. MATLAB. `http://www.mathworks.com/products/matlab/`. Retrieved at 14.03.2011.

[25] Microsoft Corporation. General information about pixel formats. `http://support.microsoft.com/kb/294880`, 2003. Retrieved at 11.03.2011.

[26] Microsoft Corporation. `http://www.windows-vista-tips-and-tricks.com/vista-and-powerpoint-2007.html`, September 2008. Retrieved at 30.05.2011.

[27] M. Narroschke and J. Ostermann. Adaptive prediction error coding in spatial and frequency domain for H.264/AVC. *ITU-T Q.6/GS16,doc. VCEG-AB06*, January 2006.

[28] G. Nattress. Chroma sampling: An investigation. `http://www.lafcpug.org/Tutorials/basic_chroma_sample.html`, July 2005. Retrieved at 11.03.2011.

[29] C. Poynton. Chroma subsampling notation. `http://www.poynton.com/PDFs/Chroma_subsampling_notation.pdf`, 2008. Retrieved at 11.03.2011.

[30] J. G. Proakis and D. G. Manolakis. *Digital signal processing. Principles, algorithms, and applications*. Prentice Hall, fourth edition, 2007.

[31] R. Radaelli-Sanchez and R. Baraniuk. Gibbs's phenomena. `http://cnx.org/content/m10092/latest/`, July 2010. Retrieved at 24.05.2011.

[32] I. Richardson. Inter prediction. `http://www.vcodex.com/files/h264_interpred.pdf`, April 2003. Retrieved at 10.05.2011.

[33] I. Richardson. Intra prediction. `http://www.vcodex.com/files/h264_intrapred.pdf`, April 2003. Retrieved at 10.05.2011.

[34] G. Ridinò. MATLAB Huffman toolbox. `http://www.mathworks.com/matlabcentral/fileexchange/4900-huffman-code`, July 2004. Retrieved at 15.03.2011.

[35] A. Said and A. Drukarev. Simplified segmentation for compound image compression. *Proceeding of ICIP*, pages 229–233, 1999.

[36] D. Salomon. *Data Compression: The Complete Reference*. Springer, second edition, 2000.

[37] K. Shring. H.264/AVC reference software. `http://iphome.hhi.de/suehring/tml/download/`. Retrieved at 11.01.2011.

[38] Y. Song. Introduction to H.264/AVC. `http://www2.engr.arizona.edu/~yangsong/h264.htm`, August 2009. Retrieved at 24.05.2011.

[39] N. Sprljan. MATLAB YUV toolbox. `http://www.sprljan.com/nikola/matlab/yuv.html`, 2011. Retrieved at 15.03.2011.

[40] Sunray Image. YUVTools. `http://www.sunrayimage.com/yuvtools.html`, 2009. Retrieved at 20.01.2011.

[41] N. G. Thompson. `http://thompsonng.blogspot.com/2010_11_01_archive.html`, November 2010. Retrieved at 24.05.2011.

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

[43] Wikipedia, The Free Encyclopedia. The gravestone of J.P.G. `http://upload.wikimedia.org/wikipedia/commons/3/38/JPEG_example_JPG_RIP_010.jpg`, September 2006. Retrieved at 07.03.2011.