SCALABLE LIVE VIDEO IN MAX/MSP/JITTER

by

Xiaonan Ma
Bachelor of Electrical Engineering, University of Ottawa

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In the
School of Engineering Science

SIMON FRASER UNIVERSITY

Fall 2010

APPROVAL

Name:	**Xiaonan Ma**
Degree:	**Master of Applied Science**
Title of Thesis:	**Scalable live video in Max/MSP/Jitter**

Examining Committee:

Chair: **Dr. Andrew Rawicz**
Professor, School of Engineering Science

Dr. Ivan V. Bajić
Senior Supervisor
Assistant Professor, School of Engineering Science

Dr. Jie Liang
Supervisor
Associate Professor, School of Engineering Science

Dr. Henry Daniel
Examiner
Associate Professor, School for the Contemporary Arts

Date Defended/Approved: 2010/10/28

ABSTRACT

This thesis describes the **mcl.jit** software library we developed to support scalable live video coding and transmission in Max/MSP/Jitter. Video codecs from this library have been successfully used in several telematic dance performances created by dancers and media artists from the School for the Contemporary Arts at Simon Fraser University during the last two years. The **mcl.jit** library also includes Region-Of-Interest (ROI) coding and motion detection objects, which can be used in a variety of interactive multimedia applications besides distributed dance performance.

We also developed a combined bit rate and frame rate control method for live video for the **mcl.jit** library. This method differs from previously developed frame rate control approaches in that it does not assume that video is pre-recorded before frame rate adjustment. The proposed method was compared to another state-of-the-art method through an extensive subjective evaluation study, the results of which indicate the superiority of the proposed approach.

Keywords: Scalable video coding; Live video streaming; Max/MSP/Jitter; Region of Interest; SPIHT; Motion tracking; Rate control.

ACKNOWLEDGEMENTS

First, I would like to thank my Senior Supervisor, Dr. Ivan V. Bajić, for his help and support during my research studies. I would also like to thank my Co-Supervisor Dr. Jie Liang, and the Examiner Dr. Henry Daniel from the School for the Contemporary Arts, who have helped me a lot during this project. I have gained considerable experience while working with them in the last two years.

I would also like to thank Dr. Andrew Rawicz for being the Chair of my M.A.Sc thesis defence. Many thanks to my examining committee for taking time to read my thesis and provide helpful comments.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1: INTRODUCTION

1.1 Motivation

With the development of broadband networking technology, distributed performance has become very popular in the new media and performing arts community. For example, a telematic dance/media performance series called Urban Fabric [5] involved one group of dancers located in Beijing, China and the other group in south California, United States. Dancers at both locations were watching the live video of each other and dancing interactively to the audience at both sites. However, this kind of telepresence performance (also known as telematic performance) requires gigabit bandwidth, and most traditional performance venues are not equipped with such high-speed access to the Internet. Making telepresence performance available to more venues and wider audience, video compression and related technologies are required to support live video transmission using the available (limited) bandwidth. To exemplify, consider uncompressed VGA (640×480) RGB video transmission at 30 frames per second (fps). Such video consumes

$$640 \times 480 \times 3 \text{ (RGB)} \times 8 \text{ bits/pixel} \times 30 \text{ fps} = 221{,}184{,}000 \text{ bits per second,}$$

that is, over 200 Megabits per second (Mbps). In other words, a gigabit network is required even for one-way transmission of uncompressed VGA colour video at 30 fps. Even the conventional 10/100 Mbps LAN would not have enough bandwidth for such video transmission. Since gigabit bandwidth is not available in most traditional performance venues, video compression must be used in these cases.

1.2 Challenges and Approaches

Worldwide network connectivity has sparked a revolution in the world of performing arts. Through the use of new communication technologies, artists want to explore new levels of thought and new types of human interaction that are not limited only to public spaces and traditional performance venues. To support creation of new media performances and installations, various software applications have been developed. Among the most widely used ones are Isadora [11] and Max/MSP/Jitter [12], as well as applications built on top of them, such as Active Space [13] and Kenaxis [14].

Several experimental performance studies using these and other similar software tools are described in [7-10]. In these studies, artists were experimenting with different types of distributed performances over an Internet connection. In most cases, large bandwidth was needed to facilitate live audio and video transmission, and various problems related to end-to-end latency, synchronization, and poor audio/video quality were encountered. For example, a third-party software "CU-SeeMe" was used in the performance described in [7] to transmit live video and audio, while running the Active Space [13] applications at the same time. The quality of the video delivered by CU-SeeMe software was relatively poor, as shown in the chat window in [7]. The network connection used in the performance described in [8] is the next-generation Internet called Interent2, which provides higher bandwidth to carry video and audio. Most traditional performance venues, however, do not have access to Internet2. The musicians in [9] were experimenting with a distributed platform for musical performance over the Internet, and they encountered problems with low audio quality and difficulties with synchronizing audio streams from different locations. The distributed musical rehearsal environment described in [10] supports both audio and video transmission between the sites. The connection between the sites, one in Germany, the other in Switzerland, was a dedicated ATM network with bandwidth of 24 Mbps. Hardware-based audio (DAT) and video (MJPEG) codecs were used. To fit the encoded PAL video into the available bandwidth of 24 Mbps and to reduce the end-to-end delay, which has a major influence on the ability of musicians to synchronize, only even fields of an interlaced PAL video were encoded. Motion-compensated video codecs were not used, since they were judged to have large encoding delay, which would have negatively impacted their application.

This last example illustrates an important trade-off related to video in distributed performances: on the one hand, motion-compensated video encoders, which are able to produce highly compressed bit streams, usually introduce too much complexity and delay to be useful in interactive performances. On the other hand, intra-frame encoders, which normally have a lower complexity and encoding delay, are not able to compress the video enough to fit into the bandwidth available in performance venues, which is usually no more than 1 Mbps. Hence, in order to have both high quality and low-delay video in a distributed performance, large bandwidth is required.

Although there are many kinds of video compression systems used in the multimedia and broadcasting industry, none of these have thus far been widely adopted by media artists. For a video codec to be useful and handy for performers and artists, it needs to seamlessly interface with (at least) one of the popular software environments commonly used in the media arts community, for example Isadora or Max/MSP/Jitter. Both Isadora and Max/MSP/Jitter provide interactive control for real-time on-stage audio and video manipulation. For example, in the performances listed in [1-4], Isadora was used to produce special patterns of light for projection on the stage, and Max/MSP/Jitter was used for live audio and video transmission.

During the course of this project, we developed a set of scalable video codecs for Max/MSP/Jitter. Here, scalability means that videos of various qualities can be decoded from a single compressed bit stream depending on the bandwidth available to a particular decoder. This kind of scalability could enable the audience with high access bandwidth to watch a less compressed (higher quality) live video, while the audience with low bandwidth receives a more compressed (lower quality) video. All the developed codecs are intra-frame, meaning that each video frame is compressed as a single image separate from other frames. Although this kind of compression has lower coding efficiency than inter-frame coding, it also has lower complexity and enables real-time operation, which is crucial for live performance. Further, intra-frame coding is more error resilient, since an error in one frame will not spread to other frames.

1.3 Max/MSP/Jitter

This section provides a brief overview of Max/MSP/Jitter [12], a graphical programming environment for music, audio, video, and data processing. Max/MSP/Jitter is a very popular software tool used for over twenty years by performers, media artists, and researchers in the fields of multimedia and computer vision. As the name suggests, the environment consists of three parts: Max, MSP, and Jitter.

Max provides user interface, timing, communications, and MIDI support. MSP is used for real-time audio synthesis and digital signal processing. Jitter, the main environment used in this work, extends Max/MSP to support real-time manipulation of video frame matrices. We provide a step-by-step tutorial on developing external objects for Jitter in Appendices 1-3, on both Mac OS X and Windows systems. The documentation and the Software Development Kit (SDK) for Max/MSP/Jitter can be downloaded from [12].

1.3.1 Max/MSP/Jitter Overview

In Max/MSP/Jitter, programs are called "patches." Each patch consists of objects connected to each other (Figure 1.1), and each object has its specific function(s). For details of its functions and usages, the user can unlock the patch and then right click on the object to open its help file.

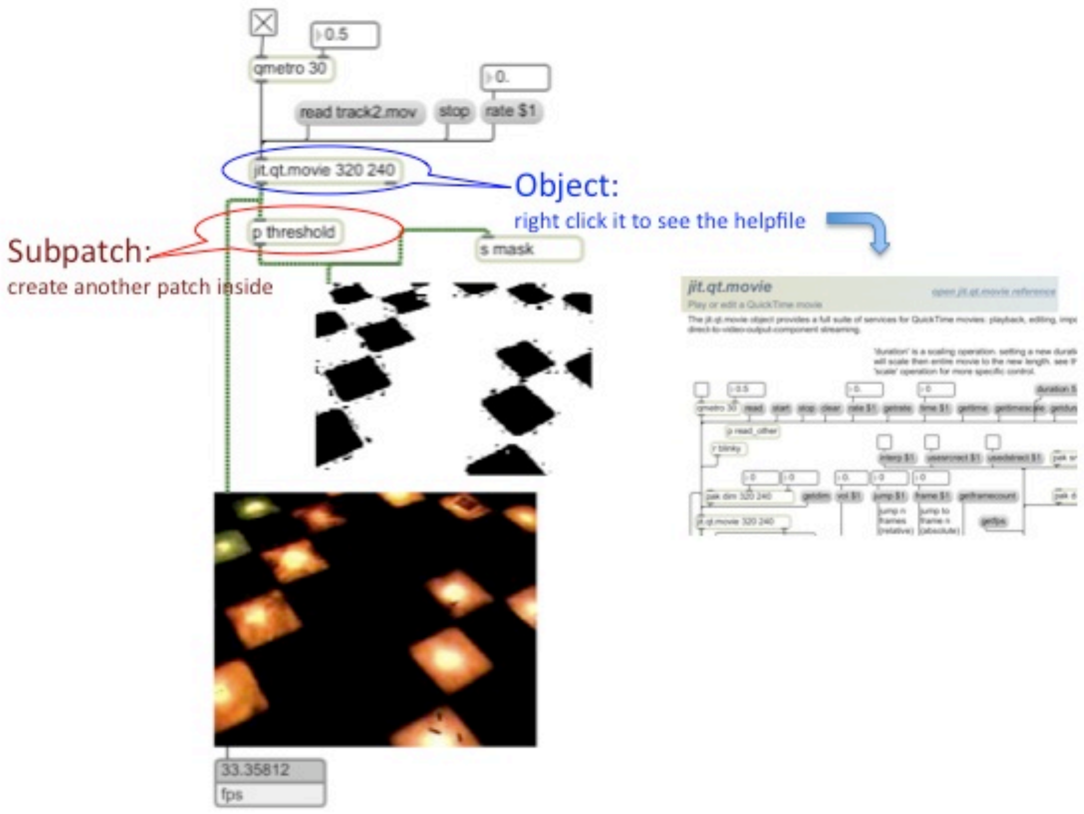


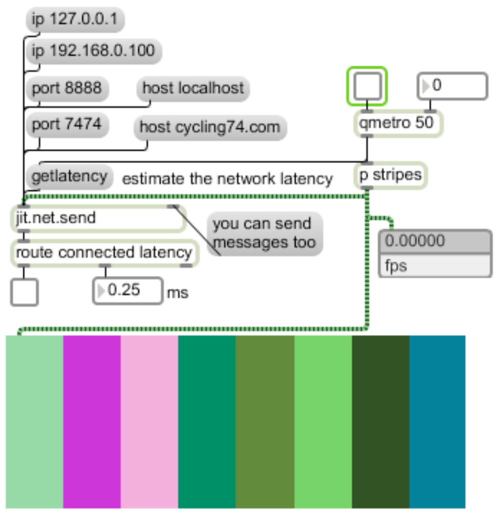
Figure 1.1: An example of a Jitter patch

As shown in Figure 1.1, objects are simply connected by chords: black chords for simple data types (numbers or text), green chords for video frames. Sub-patches can be easily created by inserting a new object and typing "**p patchname**". All data in Jitter are abstracted as multidimensional matrices. Data interchange among various objects in a patch is synchronized by an internal clock, which can be set by the **qmetro** object at millisecond precision.

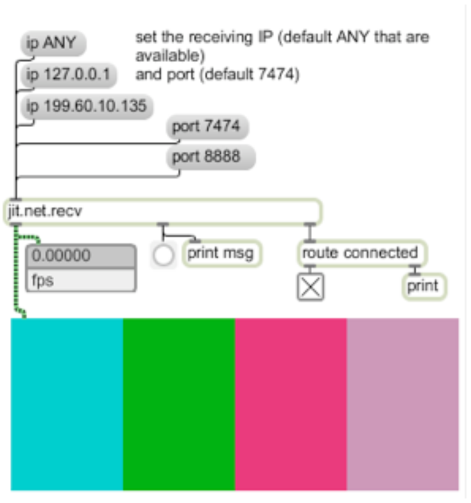
Max/MSP/Jitter is very popular in the new media arts community, due to its graphical interface and intuitive way of programming by connecting various objects. Meanwhile, it is possible to write efficient code for Jitter external objects in C/C++. This is the approach we have taken in developing scalable codecs in the **mcl.jit** library, and is described in detail in Appendices 1-3.

1.3.2 Jitter Networking

Our **mcl.jit** object library is developed to support live video streaming in Max/MSP/Jitter. In the Jitter programming environment, network communication mostly relies on two standard objects: **jit.net.send** and **jit.net.recv**. The **jit.net.send** object enables sending uncompressed Jitter matrices over an IP network to a **jit.net.recv** object running on a different computer. These two objects communicate using the TCP protocol. As shown in Figure 1.2, **jit.net.send** and **jit.net.recv** form a communication link over an IP network. The **jit.net.send** object needs to know the IP address of the receiving computer, and both **jit.net.send** and **jit.net.recv** have to listen to the same port number.



(a) jit.net.send



(b) jit.net.recv

Figure 1.2: Jitter networking patches

In Figure 1.2(a), the **qmetro** object triggers video frames every 50 milliseconds from the "p stripes" object and then forwards them to the **jit.net.send** object. The **qmetro** object also triggers a **getlatency** message. This message instructs the **jit.net.send** object to estimate one-way latency and output the estimated value from the dump outlet, which in this case was estimated at 0.25 milliseconds. The received video frames are shown in Figure 1.2(b). Note that the displayed frame here is different from Figure 1.2(a), because screenshots of these two patches were taken at different times.

1.3.3 Compiling Jitter External Objects

All external objects for Max/MSP/Jitter with "**.mxo**" extension are compiled for Mac OS. We will use an example of compiling a SPIHT encoder [16], whose source code is in C++, with Xcode 2.4.1 (Mac OS X 10.4.11) to generate an external object **mcl.jit.spihtaritenc** for Max 5. A detailed step-by-step instructions for compiling "**.mxo**" objects are presented in Appendix 1.

All external Max/MSP/Jitter objects with extension "**.mxe**" are developed for Windows. The steps to compile a Jitter external object for Windows are somewhat different from those for the Mac OS X system. The "**.mxe**" external objects in our **mcl.jit** library were all compiled in Visual C++ 2008 on a MacBook computer running Windows XP using bootcamp. In Appendix 2, we describe how to compile external Jitter objects under Windows using Microsoft Visual C++.

Compiling Jitter external objects is only a small part of the development process. A very important aspect is interfacing your code with the Jitter environment, which enables passing data such as Jitter matrices and messages between the main Max/MSP/Jitter environment and the external object. Instructions on how to interface one's code with the Jitter environment are presented in Appendix 3.

1.4 Summary of Contributions

1.4.1 Performances

The tools from our **mcl.jit** library were used in several new media dance performances listed in the table below.

Performance	Premiere venue and date	mcl.jit tools used
T2	Scotiabank Dance Center, Vancouver, July 2009	SPIHT Encoder/Decoder
T2: Echo	Emily Carr University of Arts and Design, October 2009	Frame buffer with adjustable delay
Imprint	Museum of Anthropology at UBC, January 2010	Motion detection
Imprint II	SFU Woodward's June 2010	Video/Audio Streaming

Table 1.1: List of distributed performances supported by mcl.jit tools

1.4.1.1 T2

A telematic dance performance called T2 [1], was premiered in Vancouver in July 2009. Two snapshots from this performance are shown in Figure 1.1. This performance involved two groups of dancers, one located at the Scotiabank Dance Centre in Vancouver, and another located at a gallery in Vancouver Downtown Eastside. In the middle of the performance, one dancer was transported from the Dance Centre to the gallery by a car, and a live video feed from the car was streamed to the audience at the Dance Centre (see the screen image in the right part of Figure 1.1). The audience watched that dancer leave the Dance Centre, and then saw the street view she observed from the car while travelling to the gallery. The live video streamed from the moving car was transmitted via a 3G mobile Internet link. When she arrived at the gallery, the dancer started an interactive dance (as shown in left part of Figure 1.3) with dancers at the Dance Centre. Objects from our **mcl.jit** library carried out all video compression and transmission during the performance. It is important to mention that both the gallery and the Dance Centre only had a residential-type Internet connection with a bandwidth of a few hundred kilobits per second (kbps), while the video from the moving car was transmitted via a 3G mobile Internet connection using a conventional USB 3G modem with even lower bandwidth. To our knowledge, T2 was the first ever dance performance that involved the use of live video from a moving car in telematic dance.

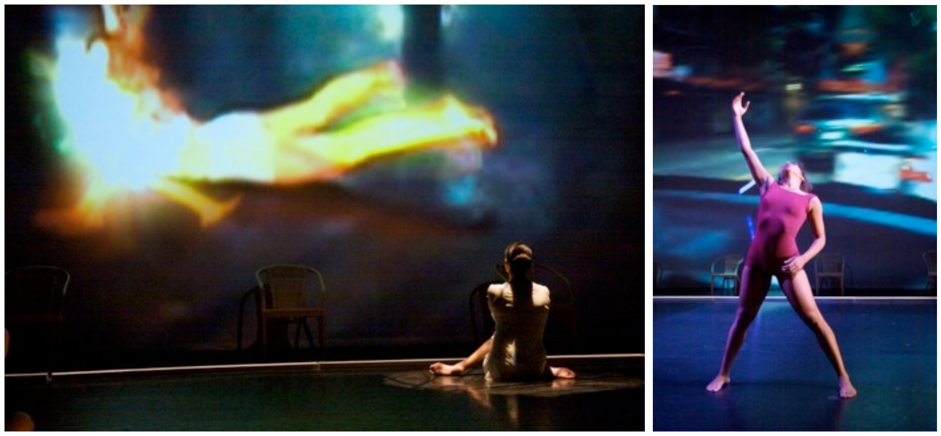


Figure 1.3: T2 dance performance: interactive dance performance (left); live street view from a moving car projected on the main stage behind the dancer (right).

1.4.1.2 T2: Echo

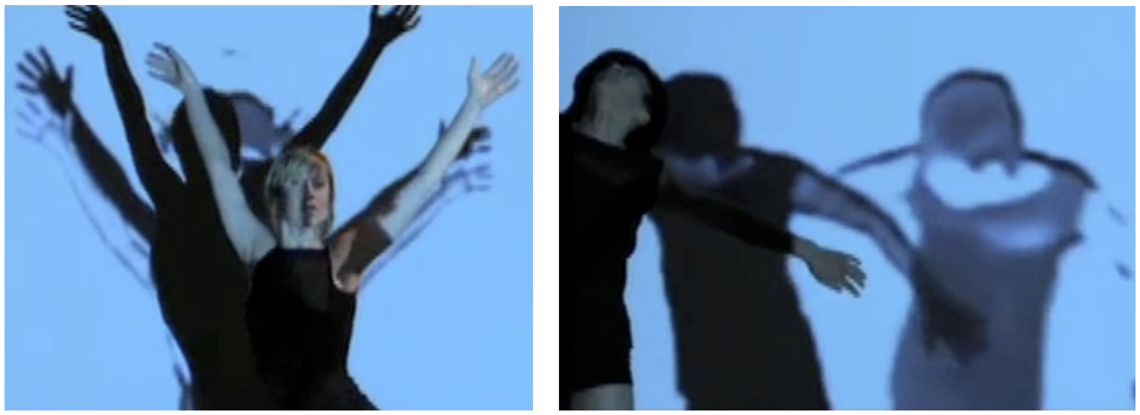


Figure 1.4: Snapshots from T2: Echo

T2: Echo [2] was performed at the Emily Carr University of Arts and Design (ECUAD) in Vancouver in November 2009. In this performance, two dancers were located at the Motion Caption (MoCap) studio at the upper floor of the main ECUAD building, and another two dancers were in the gallery at the entrance hallway of the same building. Audience were watching the dance performance at both locations. We set up a two-way live video link with adjustable delay at both ends. When a dancer was dancing in front of the camera, the audience at the other location would see his/her image projected on the background screen (Figure 1.4), mixed in with the delayed version of the same video stream and the local video stream. This created a sense of dancers interacting with their past, as well as the dancers at the other location.

1.4.1.3 Imprint

The Imprint dance performance [3] was premiered at the Museum of Anthropology (MoA) in Vancouver as part of the Vancouver 2010 Cultural Olympiad. In this performance, we have used our motion detection object

mcl.jit.motion to develop an entertainment installation for the audience before and after the show. Snapshots from Imprint are shown in Figure 1.5 below.

Figure 1.5: Imprint dance performance

1.4.1.4 Imprint II

Imprint II [4] was premiered at SFU Woodward's in June 2010 as part of the venue's opening ceremonies. This performance involved interactive dance between the Audain Gallery and the Fei and Milton Wong Experimental Theatre. Dancers in the gallery were watching the projection of the live video from the theatre, and used this visual feed to dance interactively with the dancers in the theatre. Meanwhile, the scene from the gallery was transmitted back to the theatre, and projected onto five boxes located on the stage. Since the bandwidth of the local network (which we set up ourselves, as the venue's networking infrastructure was not complete at that time) was close to 1 Gbps and therefore enough to transmit both audio and two-way video without compression, we didn't use any codecs in this performance. We only set up the two-way uncompressed video link in Max/MSP/Jitter between the two sites, as well as an audio link based on Active Space [13]. Snapshots from this performance are shown in Figure 1.6

Figure 1.6: Imprint II dance performance

1.4.2 Publications

In addition to the performances mentioned above, several technical publications resulted from our work on the development of **mcl.jit** software tools:

- I. V. Bajić and X. Ma, "A testbed and methodology for comparing live video frame rate control methods," accepted for publication in *IEEE Signal Processing Letters*, Oct. 2010.
- I. V. Bajić and X. Ma, "MCL.JIT library for scalable live video in Max/MSP/Jitter," *Proc. IEEE CCECE'10*, Calgary, AB, May 2010.
- I. V. Bajić and X. Ma, "Scalable video coding for telepresence in the performing arts," *IEEE ComSoc MMTC E-Letter*, vol. 4, no. 8, pp. 28-30, Sep. 2009. (Invited paper)

1.5 Thesis Preview

The software tools we developed to support live video in Max/MSP/Jitter are made publically available as the **mcl.jit** library [21] - a collection of external objects for Max/MSP/Jitter. We have made the library available both for the Windows and the Mac OS X systems at http://www.sfu.ca/~ibajic/mcl.jit.html. The utility of the **mcl.jit** library extends beyond new media arts, since these objects can be used for research, education, and demonstration purposes in a variety of application scenarios that involve live video transmission.

The thesis is organized as follows. The last section of this chapter describes the contributions of this work, which include performances that used our software tools, as well as several publications arising from our research. In Chapter 2 we describe SPIHT (Set Partitioning In Hierarchical Trees), the algorithm upon which our scalable codecs are based. We also describe several scalable coding objects we developed for live video. A method for combined frame rate and bit rate control of live video, as well as the results of subjective testing of our combined control method, are presented in Chapter 3. Finally, conclusions and ideas for future work are presented in Chapter 4.

2: SCALABLE VIDEO CODING USING SPIHT

Scalable video codecs in our **mcl.jit** object library are based on the SPIHT (Set Partitioning in Hierarchical Trees) algorithm described in [16] and [17]. To the best of our knowledge, this is the first scalable video codec developed as an external object for Max/MSP/Jitter. SPIHT supports quality scalability, which means that the quality of the encoded video can be easily adjusted according to user preferences or other system parameters, while various video qualities can be extracted from the same bit stream.

2.1 Versions of SPIHT

SPIHT produces an embedded (also known as progressive, or quality scalable) bit stream, which means that lower-quality versions of the image are embedded within the higher-quality versions. Hence, the more bits are decoded from the SPIHT bit stream, the better the quality of the decoded image, as illustrated in the Figure 2.1 below. Two versions of SPIHT were described in the original SPIHT paper [16], both of which produce quality-scalable bit streams:

1) SPIHT using binary coding

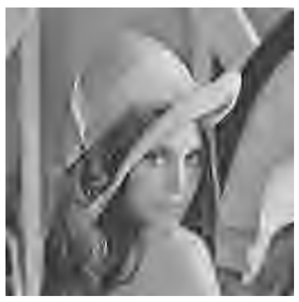
SPIHT binary codec outputs the binary code produced by set partitioning and refinement within the core SPIHT encoding procedure [16]. While this is not a perfect entropy code and its bits are still somewhat dependent, it is very fast, and results in a reasonable compression performance. If better compression performance is desired, one could follow up the binary SPIHT code by a more conventional entropy coder, such as arithmetic coder.

2) SPIHT using arithmetic coding

In arithmetic coding [18], the entire sequence of input symbols is assigned a unique binary string, which can be computed incrementally from the input data. Like Huffman coding, arithmetic coding is also asymptotically optimal. But unlike Huffman coding, arithmetic coding can easily be made adaptive to the input statistics. This is one of the reasons why it has become popular in the recent image and video coding standards, such as JPEG2000 [23], H.264 [63] and JBIG.

When arithmetic coding is used to further compress the binary code arising from the core SPIHT procedure, the overall compression performance is improved. For example, the Peak Signal-to-Noise Ratio (PSNR) of decoded images will typically increase by 0.3 to 0.6 dB for the same bit rate or file size. This approach can provide good image quality at very low bit rates, but it is more complex than the approach without arithmetic coding. As will be seen shortly, on a typical CPU, SPIHT encoder using arithmetic coding takes about twice as much time as the SPIHT encoder without arithmetic coding to encode the same image.

↓ ↓



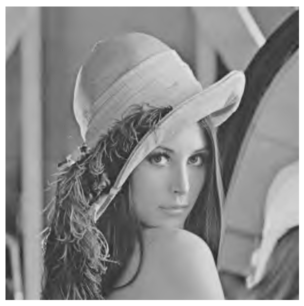


Figure 2.1: Decoding the embedded bit stream produced by SPIHT: the more bits are decoded, the better the resulting image quality

2.2 Speed and Compression Efficiency of SPIHT

The following SPIHT encoding and decoding speed (Table 2.1) and PSNR comparison (Table 2.2) were obtained on the RGB *Lena* image (512×512) at 0.5 bit per pixel (bpp). Tests were performed on a Mac Pro (Mac OS X 10.5.8 with 2×2.8GHz processor and 4 GB RAM) with standalone encoder and decoder, i.e., outside of the Max/MSP/Jitter environment. In this case, encoding and decoding time includes disk access (to read/write the raw image and/or bit stream), which is not needed in live video coding. As seen in Table 2.1 (and also in [16]), encoder is a little slower than the decoder because it requires additional operations, such as finding the largest subband/wavelet coefficient in the image.

Bitrate=0.5bpp	**Arithmetic**	**Binary**

Encoder	0.05 sec	0.05 sec
Decoder	0.03 sec	0.02 sec

Table 2.1: SPIHT encode/ decode speed on a Mac Pro

Bitrate=0.5bpp	**Arithmetic**	**Binary**
PSNR	32.42 dB	32.06 dB

Table 2.2: PSNR comparison

(a) The original *Lena* image

(b) Arithmetic, PSNR=32.42 dB

(c) Binary, PSNR=32.06 dB

Figure 2.2: Original (a) and decoded *Lena* image using (b) arithmetic and (c) binary SPIHT coding at 0.5 bpp.

SPIHT with arithmetic coding has higher compression efficiency and provides higher quality (up to 0.5 dB in PSNR at a fixed bit rate) than the codec without arithmetic coding, but with a 2-2.5 times higher computational cost in terms of CPU time [16]. However, in practical live video applications for which our codec library is developed, the total processing time is dominated by memory access, not the CPU time. In the test above, it appears that the disk access was an important factor for the overall speed (Table 2.1). In our experiments with live video, which will be discussed in more detail later in this chapter, we found that codecs using arithmetic coding were only about 10-20% slower than the corresponding codecs without arithmetic coding.

2.3 SPIHT External Objects for Max/MSP/Jitter

The **mcl.jit.spihtaritenc** is the external object we created for encoding video frames using SPIHT with arithmetic coding. It takes live video from **jit.qt.grab** on Max OS (or **jit.dx.grab** on Windows), produces the compressed bit stream, and casts it as a Jitter matrix data structure. In addition to the video frames, the inputs to the encoder are the encoding bit rate in bits per pixel (bpp) and the dimensions of the frame in pixels. The **mcl.jit.spihtaritdec** is the corresponding decoder, which takes the compressed bit stream in the Jitter matrix and decodes the video frame. In addition to the compressed bit stream, the input to the decoder is the decoding bit rate. We have also created the encoder/decoder pair based on SPIHT with binary coding, which are called **mcl.jit.spihtbinenc** and **mcl.jit.spihtbindec.**

2.3.1 SPIHT with Region Of Interest (ROI) Coding

Region-Of-Interest (ROI) coding is a simple feature we added to SPIHT codecs in the **mcl.jit** library. The corresponding ROI-capable SPIHT codecs are

mcl.jit.spihtaritROIenc/dec, and **mcl.jit.spihtbinROIenc/dec.** In addition to the video frame, these codecs expect as inputs the upper left and lower right coordinates of a rectangular ROI, which will be encoded at a higher quality than the rest of the frame. The higher coding quality is achieved by up-shifting the bit planes of the subband/wavelet samples within ROI relative to the rest of the frame. This method is essentially the same as ROI coding by scaling in JPEG2000 [23].

The idea behind ROI coding by bit plane shifting is shown in Figure 2.3. The highlighted square S_k is the set of sample coordinates within a subband at level k, which corresponds to the Region Of Interest (ROI). In our ROI coding, all subband samples at level k whose coordinates are inside the highlighted square S_k will be multiplied by 2^U, where the integer U is the so-called "up-shift" factor [23]. Multiplication by 2^U will result in up-shifting of the bit planes of the samples inside S_k. Because SPIHT coding proceeds from the most significant bit plane towards less significant bit planes, the sample values from ROI will appear earlier in the compressed bit stream relative to the samples of the same significance (with respect to a given threshold) from the non-ROI part of the frame. This means that at any bit rate, ROI samples will effectively be quantized with a finer quantizer, and therefore have better quality. The decoder needs to know the value of U and the location of the ROI in order to perform reverse operations. For simplicity, our current implementation supports only one rectangular ROI, so we only need to transmit the coordinates of upper left and lower right corner of the ROI rectangle to the decoder and the value of the up-shift factor U. These values are transmitted as header information in the bit stream. An example of using

SPIHT with ROI is given later in this chapter, where we combined ROI coding with face detection to encode the face region with higher quality.

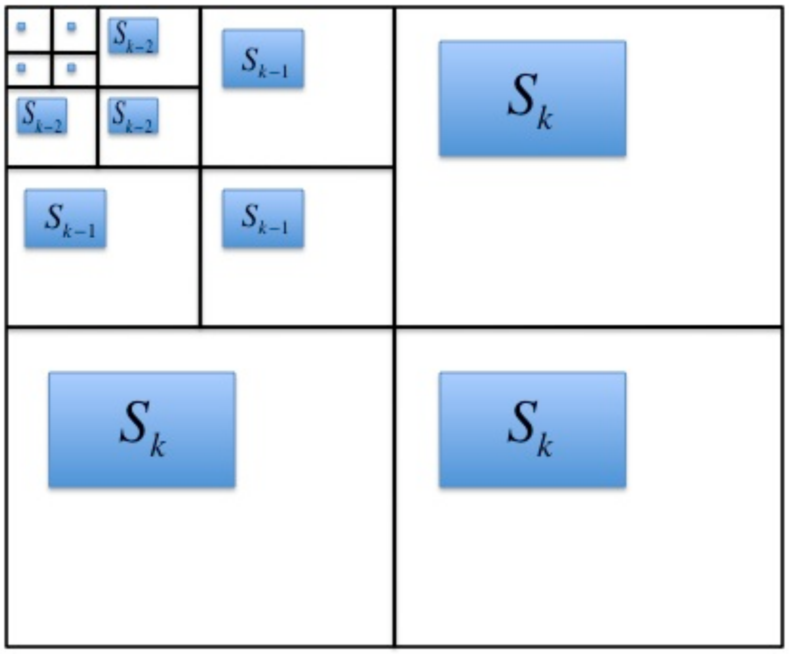


Figure 2.3: Subband/wavelet coefficients corresponding to a rectangular ROI

All SPIHT external objects in the **mcl.jit** library (both ROI and non-ROI) take 4-plane (RGB plus Alpha) video input, which is the common video format in Max/MSP/Jitter, and produce a quality-scalable bit stream cast as a Jitter matrix data structure, which can then be sent to other Jitter objects. Alpha plane is not being encoded into the bit stream; it is discarded at the encoder, and the default opaque Alpha plane is re-created at the decoder.

2.4 Performance evaluation within Max/MSP/Jitter

SPIHT with arithmetic coding has the compression performance which is comparable with JPEG2000 [23]. The reasons for choosing SPIHT over JPEG2000 are the fact that it has better (finer) quality scalability and simple bit rate control. Both characteristics are useful for Max/MSP/Jitter applications. In this section, we will focus on performance evaluation of SPIHT compression of live video within Max/MSP/Jitter. The last two sections of this chapter will demonstrate the uses of SPIHT compression in live video transmission and ROI coding.

2.4.1 Encoding speed

Speed tests for live video coding were carried out on a Mac Pro (Mac OS X 10.5.8 with 2×2.8GHz processor speed and 4 GB RAM). This computer was set up as a transmitter and was sending encoded live video (captured using a Minoru webcam) to a MacBook Pro (Mac OS X 10.5.8 at 2.5 GHz with 2 GB RAM) over a LAN. Both computers were running Max/MSP/Jitter version 5.0.8 for Mac OS X. Video compression was carried by **mcl.jit.spihtaritenc/dec** and **mcl.jit.spihtbinenc/dec** objects.

The speed test results for both 320×240 and 640×480 RGB video are shown in Figure 2.4. In this test, the transmitter was set to capture frames at 30 millisecond (ms) intervals (which corresponds to the frame rate of 30.3 fps), encode them, and send them to the receiver. When the processing time related to one frame (capturing, encoding, etc.) starts approaching 30 ms, the frames will start to get dropped, and the frame rate at the receiver will start to decrease. For the 320×240 resolution, the achieved frame rate using SPIHT with arithmetic coding remains at or above 30 fps when the encoding bit rate is less than 2.4 bpp, while the frame rate of using SPIHT with binary coding, which is computationally more efficient, stays at or above 30 fps for the entire range of tested bit rates up to 5.6 bpp. Both the encoder and the decoder require more processing as the bit rate increases, since there are more bits to compute. Hence, frame rate reduction at higher bit rates is to be expected. At the resolution of 320×240, frame quality is acceptable at bit rates 0.5 - 0.8 bpp, and very good at bit rate at 1.0 bpp (a sample decoded frame is shown in Figure. 2.5(a)), which means that one can easily achieve very good live video quality with frame rates above 30 fps at this resolution.

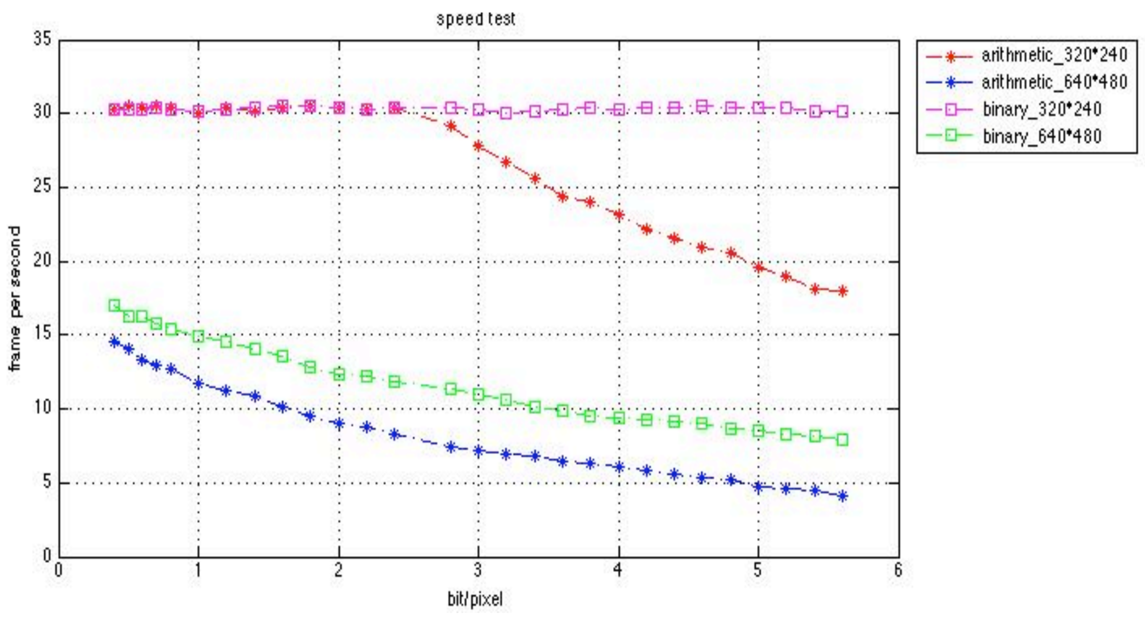


Figure 2.4: Measured frame rate vs. bit rate for live 320×240 and 640×480 RGB video.

At the resolution of 640×480 there are four times as many pixels as there were at the resolution of 320×240. Hence, we observe from the figure that the achieved frame rate for each of the two codecs (arithmetic and binary) is roughly four times lower at higher bit rates than that achievable at the 320×240 resolution.

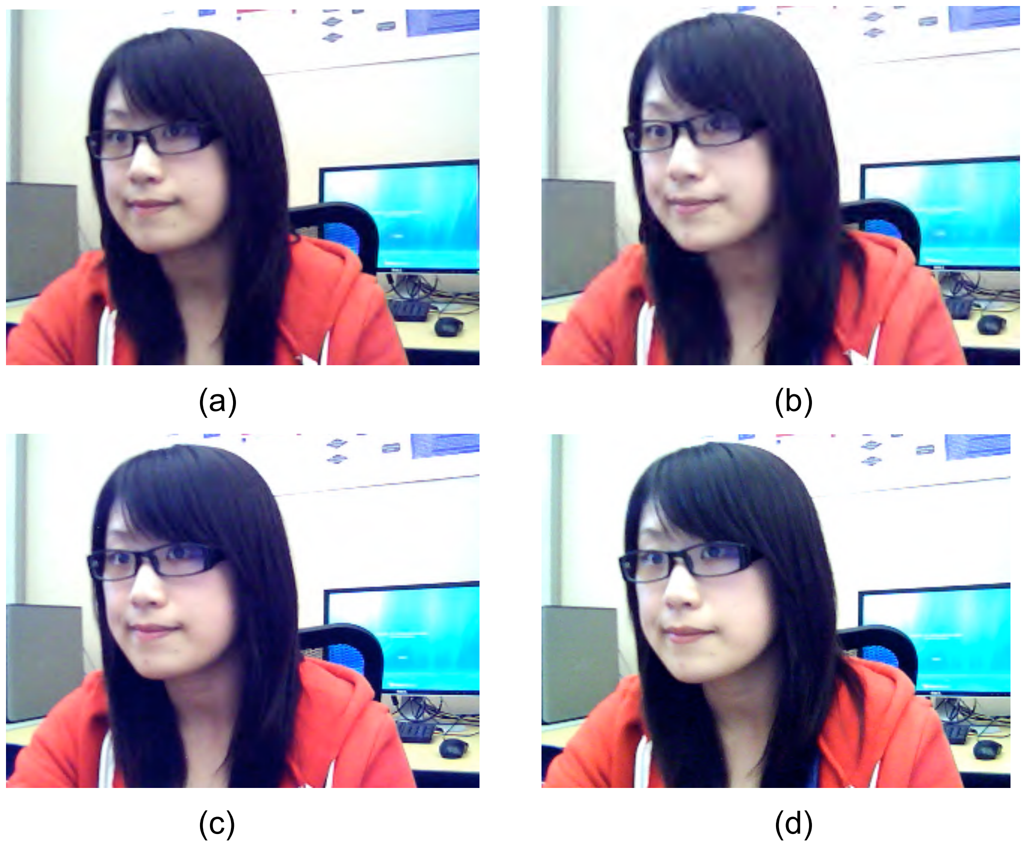


Figure 2.5: Visual quality comparison at 1.0 bpp for: (a) 320×240 frame using arithmetic encoding; (b) 320×240 frame using binary encoding; (c) 640×480 frame using arithmetic encoding; (d) 640×480 frame using binary encoding.

A comparison of decoded frame qualities is shown in Figure 2.5. The frames were taken by a Minoru webcam. For this example, the encoder and the decoder were both running on a Mac Pro system mentioned above. We used the encoding bit rate of 1.0 bpp for both arithmetic and binary SPIHT coding. The top two images show the decoded frame qualities at the 320×240 resolution, while the bottom two images show the decoded frame quality at the 640×480 resolution. As seen in the Figure 2.5, at the bit rate of 1.0 bpp, the frame quality is very good at both resolutions. Hence, we can achieve very good frame quality at 320×240 and 30 fps with either binary or arithmetic SPIHT codec, and very good quality at 640×480 and 15 fps using binary SPIHT coding. At 640×480, with the bit rate of 1.0 bpp, SPIHT with arithmetic coding was only able to provide around 12 fps on our Mac Pro system.

2.4.2 Visual delay

Delay plays an important role in visual communications. Measuring the propagation delay between two computers on an IP network is relatively easy using the “ping” command on a Unix/Linux/Windows prompt. However, measuring the “visual delay,” which is the time it takes for one video frame to get captured at one computer, then transmitted and displayed at the other computer, is more involved. This delay includes frame acquisition, encoding, transmission, decoding, and rendering. In order to measure visual delay we used the following methodology. Two computers were connected to the same 100 Mbps Ethernet switch, so that the “ping” round trip time between them was less than 1 ms. One computer (Windows XP Pentium 4 at 3.4GHz with 1 GB RAM) was acting as a transmitter. Both computers were running Max/MSP/Jitter version 5.0.8 for Windows. The transmitter was set to grab 320×240 RGB frames at 33 ms intervals, corresponding to the frame rate of about 30 fps.

Both computers’ clocks were synchronized to the same atomic clock NTP server, and the local time of each computer was displayed on its screen with millisecond precision. The webcam was then pointed to the clock on the screen of the computer it was connected to, and the video signal was grabbed into Max/MSP/Jitter and sent to the other computer, where it was displayed alongside the local clock of that computer.

A sample screenshot from these experiments is shown in Figure 2.6. In this example, which was obtained by directly transmitting captured frames without compression, there is a 292 ms difference between the two clocks, indicating that the visual delay can be significant even when the propagation delay is very small (less than 1 ms in this scenario, since the transmitter and the receiver were both connected to the same switch). The main reason for this large visual delay in these experiments is the slow sensor readout in the webcam.

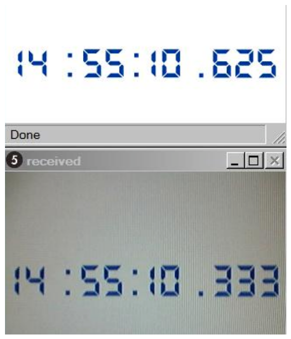


Figure 2.6: Part of the captured screen showing the local clock at the receiver (top), and the received image of the transmitter's clock (bottom), with millisecond precision

We performed 10 measurements of the visual delay without compression, i.e. with 320×240 RGB frames sent directly to the receiver, and 10 measurements with encoding at 0.5 bpp using **mcl.jit.spihtaritenc**. The average visual delay without compression was 311.9 ms (standard deviation 23.2 ms), while the average visual delay with compression at 0.5 bpp was 327.5 ms (standard deviation 22.4 ms). Hence, at this resolution and bit rate of 0.5 bpp, our **mcl.jit.spihtaritenc** codec adds, on average, less than 20 ms (i.e., less than one frame interval) to the visual delay, which is fairly low, especially considering that the codec is purely software-based. Since the binary version of the codec is faster than the arithmetic version, that codec would also add no more than one frame delay into the total visual delay.

2.5 Scalable live video transmission

Since SPIHT codecs produce quality-scalable bit streams, various qualities of displayed frames can be obtained by truncating the bit stream at various points. The output bit stream of our SPIHT encoder objects is cast as an array of 8-bit numbers forming a Jitter matrix, which means that it can be directly applied as an input to other Jitter objects.

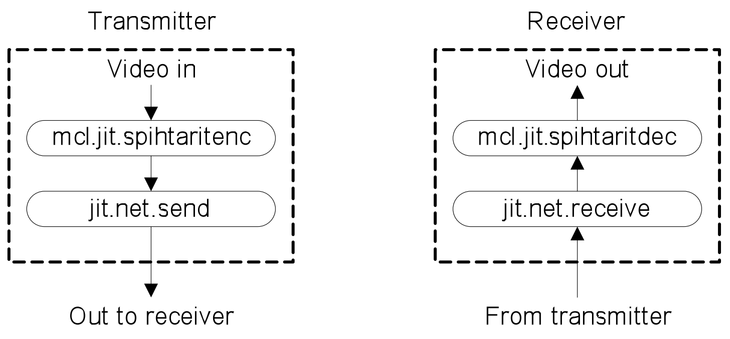


Figure 2.7: Using the mcl.jit.spihtarit codec together with jit.net.send/recv to form a video communication link

In order to set up a video transmission connection in Jitter over an IP network, one can use the existing **jit.net.send** and **jit.net.receive** objects. The SPIHT codecs can be used together with these two objects to form a video link, as illustrated in Figure 2.7. The encoding bit rate and the IP addresses for the two networking objects are omitted in this figure for simplicity. Our SPIHT encoders allow users to adjust the bit rate manually in order to provide appropriate video quality for a particular scene. The encoding bit rate can also be adjusted dynamically during transmission according to current network conditions using the TFRC method, which will be described in the next chapter.

2.5.1 Point-to-point and point-to-multipoint live video streaming

In a simple setup for point-to-point live video streaming shown in Figure 2.7, one computer captures the video, compresses it using the SPIHT encoder object **mcl.jit.spihtaritenc**, and sends it to another computer over an IP network. The receiving computer running **mcl.jit.spihtaritdec** then decodes the compressed bit stream and displays the video.

If there are multiple receivers that want to receive live video, each would run an instance of **jit.net.receive**, and then decode the compressed bit stream as shown in the right part of Figure 2.7. Meanwhile, the transmitter would encode its live video using an encoder and then send it to the receivers using one instance of **jit.net.send** for each receiver. Even if users require different bit rates, the transmitter needs to encode the video only once (i.e., produce one scalable bit stream at the highest requested bit rate), and then optionally truncate it to lower bit rates for low-bandwidth receivers. Since the bit stream is cast as a Jitter matrix, truncation can be easily accomplished using an existing object called **jit.submatrix**, which will be illustrated in the next section.

2.5.2 Peer-based live video multicast

When the bit stream scaling is performed at an intermediate peer node rather than the transmitter, we refer to that process as peer-based scaling. The basic principle of peer-based scaling is illustrated in Figure 2.8. Since the SPIHT bit stream is quality scalable, creating a bit stream at a lower bit rate simply amounts to bit stream truncation, which can be accomplished using the existing **jit.submatrix** object. Hence, a peer node does not need to decode and re-encode the video. Since the scaling operation is so simple, it can be easily and quickly executed on a conventional computer. Using this simple set up, we can construct a peer-based multicast tree, where the compressed bit stream for each end-user is scaled appropriately to its requested bit rate by the intermediate peer nodes.

In the scenario shown in Figure 2.8, the peer node is sending video to two downstream users, one with high available bandwidth (top right) and the other with low available bandwidth (bottom right). For a given frame, a scalable bit stream at rate r is received at the peer node, and the encoding bit rate of that bit stream is stored in its header. The peer node then forwards the complete bit stream to the high-bandwidth user, while a truncated version (in this example at rate r/4, i.e., one quarter of the original encoding bit rate) is sent to the low-bandwidth user. The new bit rate (r/4) is stored in the header of the bit stream sent to the low-bandwidth user.

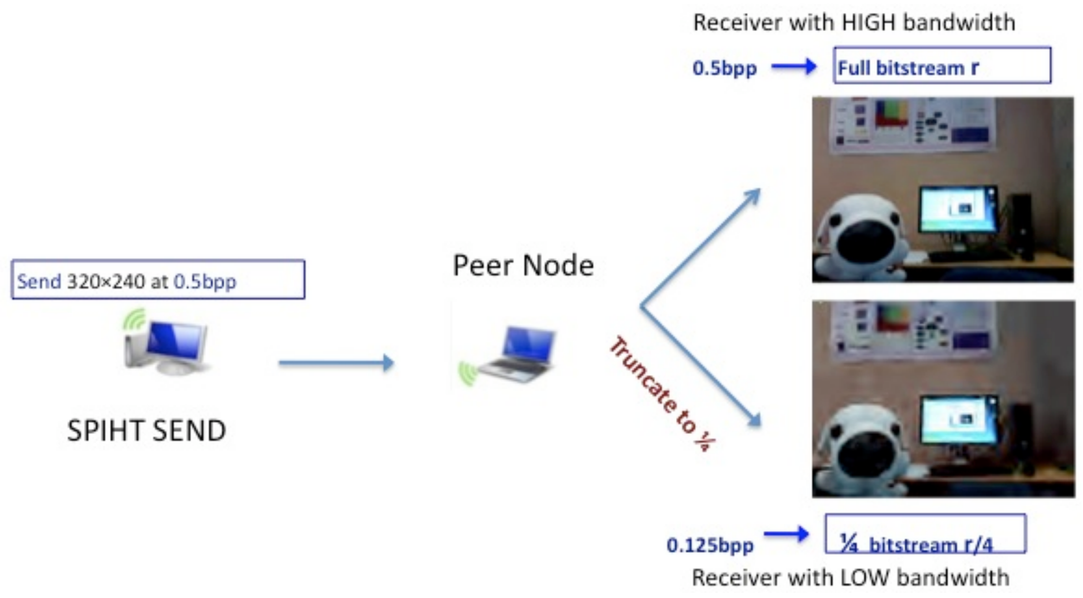


Figure 2.8: Peer-based live video multicast

To demonstrate the utility of our codecs in peer-based multicast, we set up four computers in our lab in the configuration shown in Figure 2.8. One computer (Mac Pro) captures the video, compresses it using the **mcl.jit.spihtaritenc** object, and sends it to the peer node (PC in this case) over an IP network. The peer node then adjusts the compressed bit stream to suit two receivers downstream. In this example, one of the receivers (MacBook with 2 GHz processor and 2 GB RAM) has a low-bandwidth connection, so the peer adjusts the compressed bit stream to one quarter of the encoding bit rate and sends the truncated bit stream to this receiver. The bit stream header is updated to reflect the bit rate change. The other receiver (MacBook Pro with 2.5 GHz processor and 2 GB RAM) has higher available bandwidth. The peer simply forwards the complete compressed bit stream to it, without any truncation. The complete Max/MSP/Jitter patch running at the peer node is shown in Figure 2.9. In this patch, object **jit.split** separates the encoding bit rate (which is stored in the header - the first two bytes of the bit stream) from the remainder of the bit stream, and then **jit.submatrix** truncates the bit stream. New bit rate is appended to the truncated bit stream and sent to the low-bandwidth user. Note that decoding and re-encoding is not needed in this process. Hence, this truncation process represents a very efficient form of *transcoding*, which is only possible with quality-scalable bit streams.

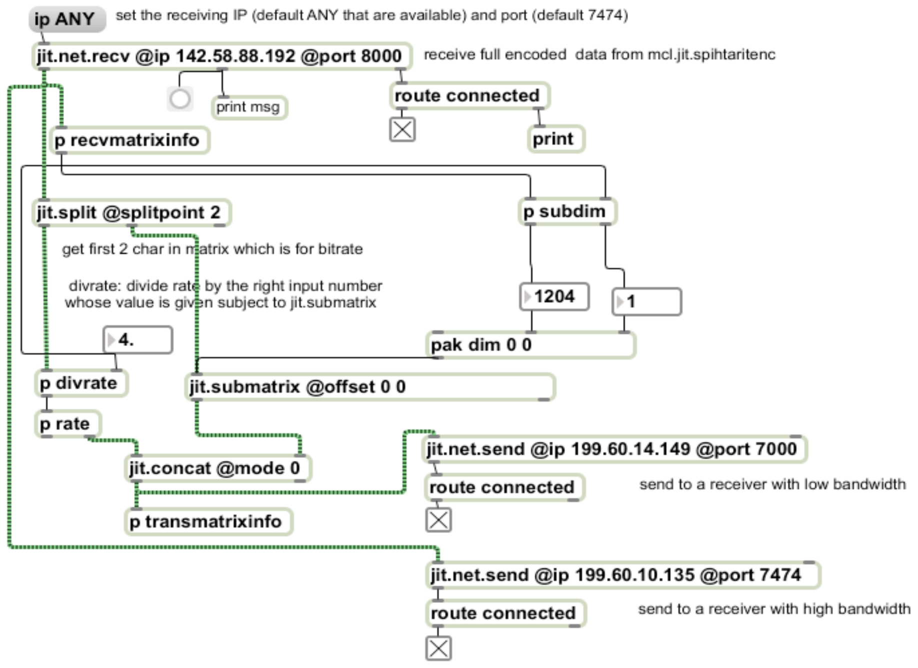
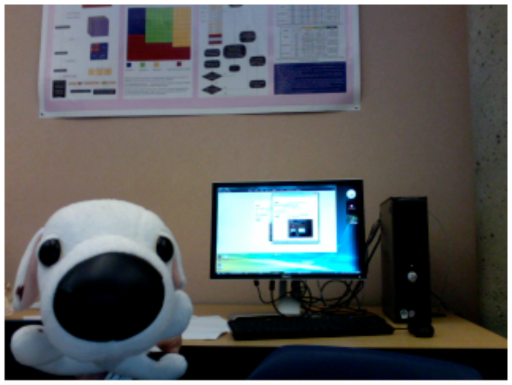
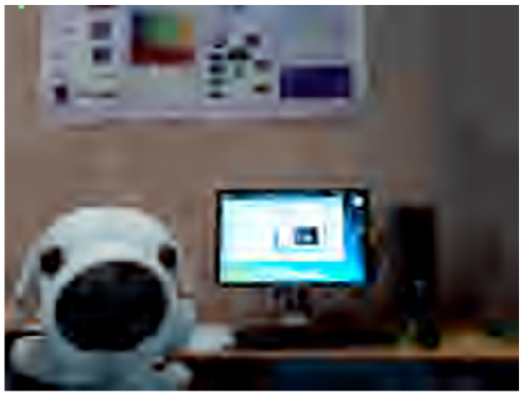

ip ANY
set the receiving IP (default ANY that are available) and port (default 7474)
jit.net.recv @ip 142.58.88.192 @port 8000
receive full encoded data from mcl.jit.spihtaritenc
print msg
route connected
print
p recvmatrixinfo
jit.split @splitpoint 2
p subdim
get first 2 char in matrix which is for bitrate
divrate: divide rate by the right input number whose value is given subject to jit.submatrix
1204
1
4.
pak dim 0 0
p divrate
jit.submatrix @offset 0 0
p rate
jit.net.send @ip 199.60.14.149 @port 7000
jit.concat @mode 0
route connected
send to a receiver with low bandwidth
p transmatrixinfo
jit.net.send @ip 199.60.10.135 @port 7474
route connected
send to a receiver with high bandwidth

Figure 2.9: Truncator patch in a peer-based multicast setup

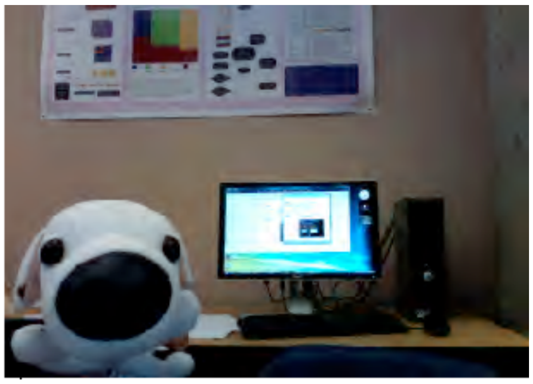
Sample frames of the original video and decoded videos at the two receivers are shown in Figure 2.10. Note that the video at the low-bandwidth user has a correspondingly lower quality than the video at the high-bandwidth user.



(a) Original video sent from Mac Pro



(b) Low bandwidth receiver (1/4 bit stream)



(c) High bandwidth receiver (full bit stream)

Figure 2.10: Quality comparison of received videos in a peer-based multicast

2.6 ROI coding demonstration

To demonstrate ROI coding, we have combined our **mcl.jit.spihtaritROI** codec with the face detector in the external Max/MSP/Jitter object **cv.jit.faces** [25], which is an implementation of the well-known Viola-Jones face detector [26]. The face detector finds the enclosing rectangle for the face (see Figure 2.11 below) in the frame and feeds the coordinates of its upper-left and lower-right corner to the ROI encoder. These coordinates are used to perform bit plane up-shifting as explained earlier. They are also stored in the header of the bit stream along with the up-shift factor U, so that the decoder can perform the reverse operation.

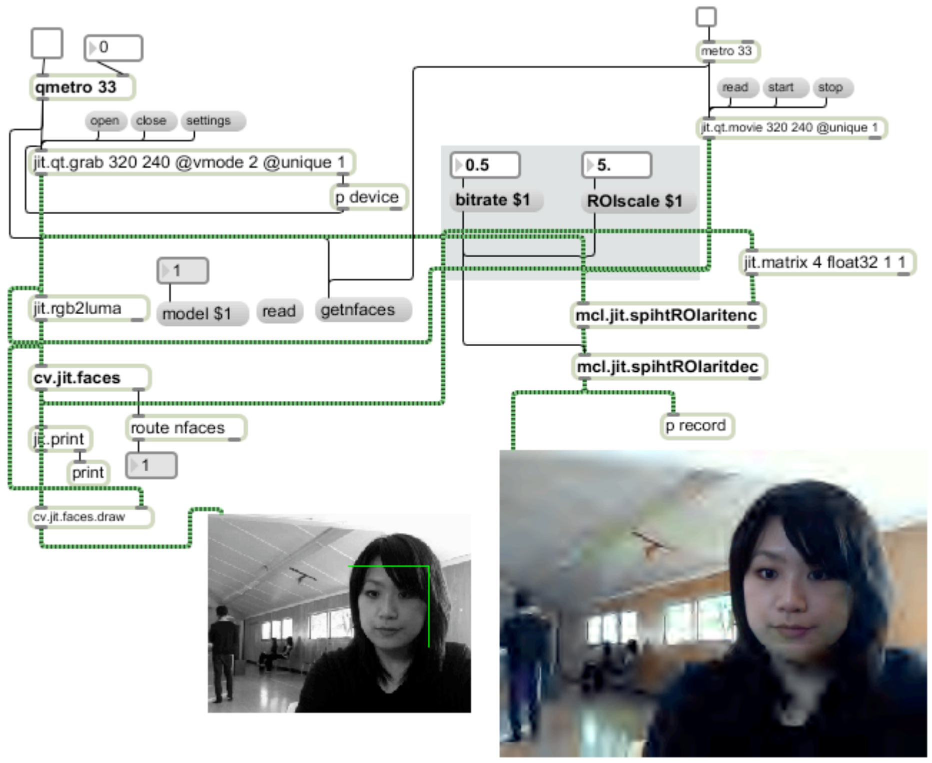


Figure 2.11: Face detection using cv.jit.faces and ROI coding at 0.5bpp with U=5

With a higher value of the up-shift factor U, the quality contrast between the face area and the rest of the frame will be larger. As shown in Figures 2.12 - 2.14, U=5 results in a very noticeable difference between the face and background, compared to up-shifting by U=1 or U=3. A relatively low bit rate of

0.3 bpp is chosen to emphasize the difference in quality between the ROI (face) and the rest of the frame in these examples.

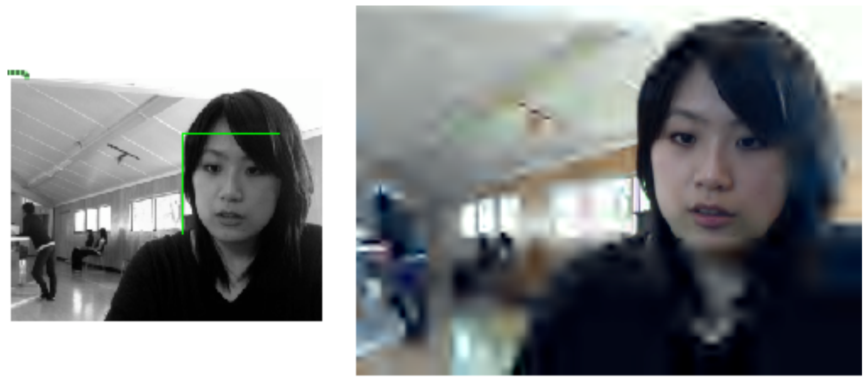


Figure 2.12: Sample ROI frame encoded at 0.3 bpp with U=5

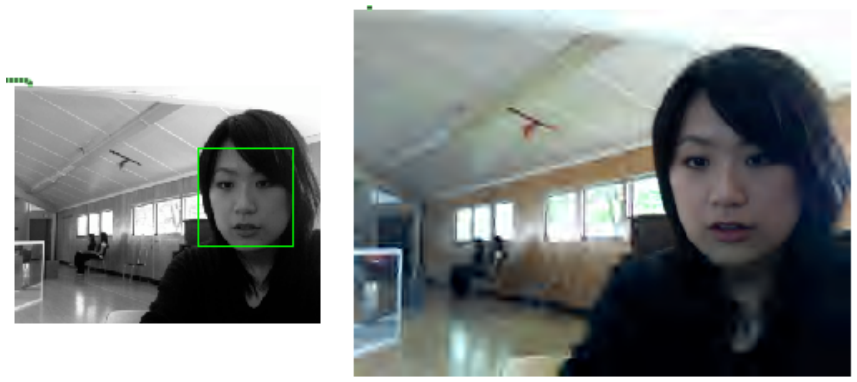


Figure 2.13: Sample ROI frame encoded at 0.3 bpp with U=3

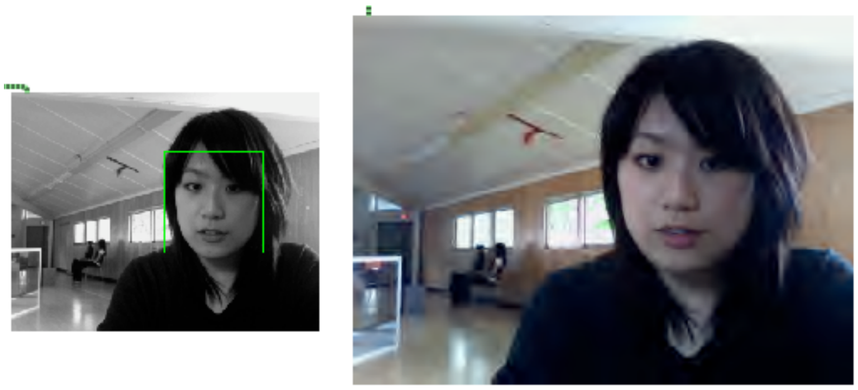


Figure 2.14: Sample ROI frame encoded at 0.3 bpp with U=1

Even though the examples above are all related to face ROI, the codec itself can be used encode any other rectangular ROI, for example a part of a moving object. All that is needed is to identify the desired region in the frame and feed the coordinates of its enclosing rectangle to the encoder. The resulting bit stream is still quality scalable: as more bits are decoded, the quality of both the ROI and the rest of the frame improves.

3: COMBINED FRAME RATE AND BIT RATE CONTROL

When dealing with live video, the ability to vary video frame rate could be very helpful. If the video scene is static, there is no need to represent it with high frame rate. On the other hand, when there is motion in the scene (for example, when the dancers enter the scene and start to move), the frame rate should be increased correspondingly to ensure motion smoothness. Several methods for frame rate control have been presented in the literature [37], [41] and [42]. The frame rate control methods in these papers are performed on pre-recorded videos. In other words, they assume that video has been already recorded and stored at some frame rate (e.g. 30 fps), and the goal is to find out which frames can be dropped from the video stream without affecting the visual quality too much. However, in the scenario we are focusing on (distributed performance), we are dealing with live (i.e., not pre-recorded) video most of the time. Therefore, the approaches proposed in [37], [41] and [42] might not be the most appropriate. Ideally, frames that will eventually be dropped should not be captured in the first place, since those frames would simply use up memory and processing power without being displayed.

There is another reason why an approach different from those in [37], [41] and [42] might be better for frame rate control. Once the frames are captured at a certain frame rate, the set of frame rates that can be obtained from such a stream by dropping frames is limited. For example, if the video is captured at 30 fps, the only possible spacing between frames is a multiple of 33.3 ms.

Therefore, the set of actual instantaneous frame rates is 30/*n* fps, where *n* is an integer. On the other hand, if we were able to fully control the frame sampling interval τ, we could achieve any instantaneous frame rate $1/\tau$ fps.

In this chapter, we propose a simple method for combined frame rate and bit rate control. This method allows easy control of the frame sampling intervals and encoding bit rate for live video streaming. The encoding bit rate control is based on TCP-Friendly Rate Control (TFRC), which is described in Section 3.1. The proposed frame rate control method was compared to the one in [37] through extensive subjective evaluation. The reason for choosing the method in [37] for comparison is that this method is the easiest to cast into the context of live video among the three methods in [37], [41] and [42]. This method, and its implementation in Max/MSP/Jitter, is described in Section 3.2. Our frame rate method is introduced in Section 3.3. Subjective evaluations of the videos produced by these two methods are described and analyzed in Sections 3.6 and 3.7.

3.1 TCP-Friendly Rate Control (TFRC)

3.1.1 Network setup in Jitter

The **jit.net.send** and **jit.net.recv** objects in Max/MSP/Jitter communicate using the TCP protocol. In TCP, reception of information at the receiver is acknowledged by sending a confirmation message to the transmitter, as shown in Fig. 3.1

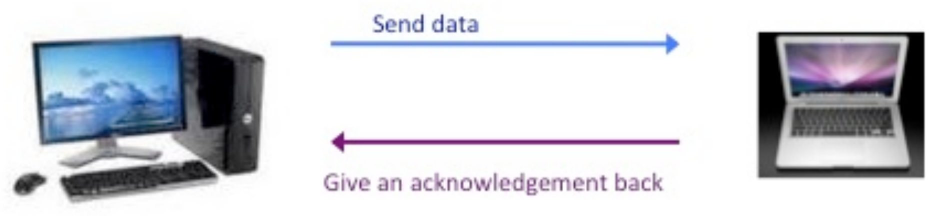
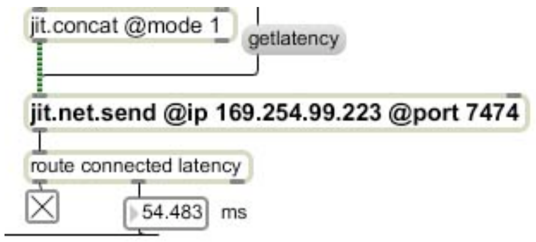


Figure 3.1: Data flow with TCP

The feedback mechanism present in the TCP connection enables the system to estimate the end-to-end delay between the transmitter and the receiver. The end-to-end delay (also known as latency) can be estimated by using a **getlatency** message connected to the input of the **jit.net.send** object, as shown in Figure 3.2. The latency is displayed in milliseconds.



```
jit.concat @mode 1    getlatency
jit.net.send @ip 169.254.99.223 @port 7474
route connected latency
54.483  ms
```

Figure 3.2: Latency estimate is output through the dump outlet

There are two factors that could influence the latency: the amount of data being sent, and the actual transmission time between two computers. For a given bandwidth (in bits/second), the more data that is sent, the longer time it would take to move all the data through the network. Transmission can become quite slow if the server keeps sending video data with both a high frame rate and a high encoding bit rate (or if the video is uncompressed). In such cases, Max window will give the error message saying that the data is being input faster than it could be sent, and the received video will look jerky and have a fairly low frame rate. In order to avoid having both high frame rate and high bit rate at the same time, the TCP-Friendly Rate Control (TFRC) is added as a rate control module to our SPIHT encoder.

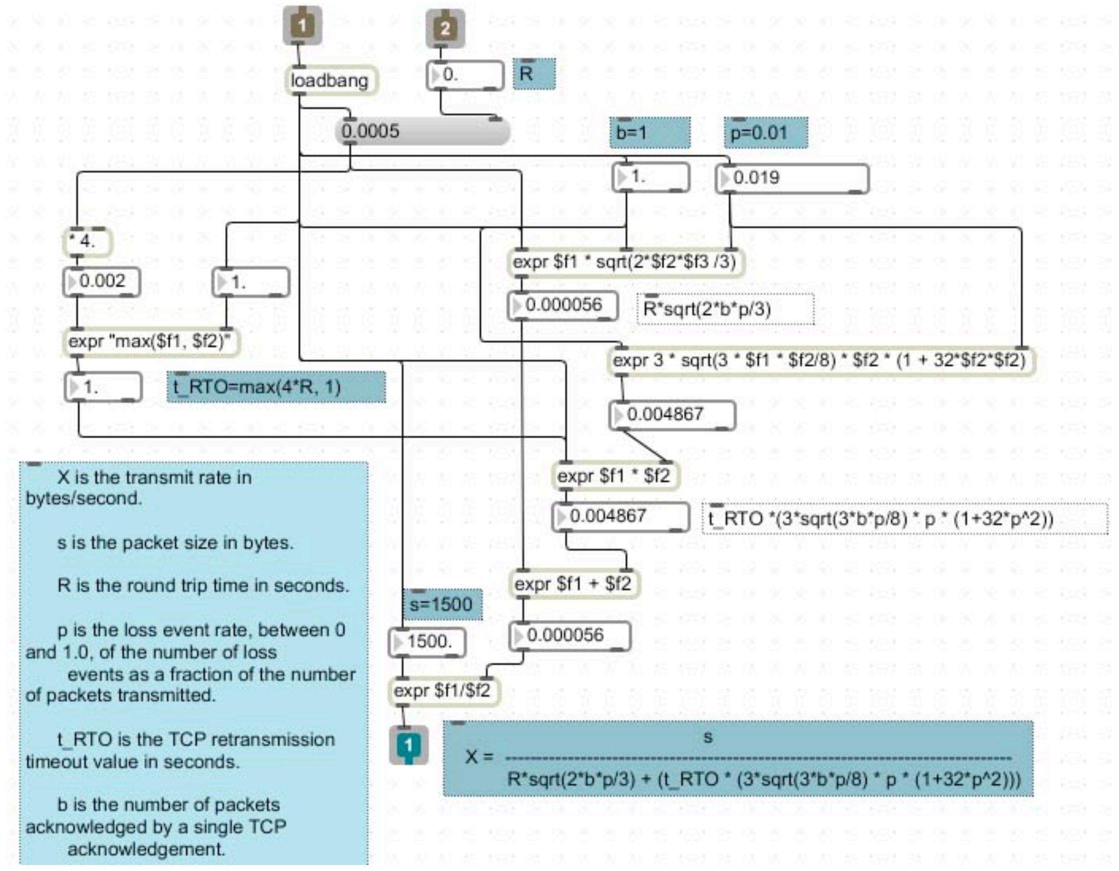
3.1.2 TCP-Friendly Rate Control

TCP-Friendly Rate Control (TFRC) proposed in [28-29] is designed to control the congestion of unicast (usually non-TCP) flows working in an Internet environment and competing for bandwidth with other TCP flows. Several applications of TFRC in video streaming have been described in the literature, for example [31-35]. These works firmly establish applicability of TFRC to video streaming.

TFRC is based on estimating the average sending rate of a TCP flow under packet loss and round trip delay as specified in the following equation:

$$X = \frac{s}{R \times \sqrt{2 \times b \times \frac{p}{3}} + t\_RTO \times 3 \times \sqrt{3 \times b \times \frac{p}{8}} \times p \times (1 + 32 \times p^2)} \tag{3}$$

where *X* is the transmission rate (in Bytes/second), *s* is the packet size (in Bytes), *R* is the Round Trip Time (RTT) in seconds, *t_RTO* is the TCP retransmission timeout value in seconds, *b* is the maximum number of packets acknowledged by a single acknowledgement packet, and *p* is the packet loss rate. Implementation of this equation in a Max/MSP/Jitter patch is shown in Figure 3.3 below.


1
2
loadbang
0.
R
0.0005
b=1
p=0.01
1.
0.019
* 4.
0.002
1.
expr $f1 * sqrt(2*$f2*$f3 /3)
0.000056
R*sqrt(2*b*p/3)
expr "max($f1, $f2)"
expr 3 * sqrt(3 * $f1 * $f2/8) * $f2 * (1 + 32*$f2*$f2)
1.
t_RTO=max(4*R, 1)
0.004867
expr $f1 * $f2
0.004867
t_RTO *(3*sqrt(3*b*p/8) * p * (1+32*p^2))
expr $f1 + $f2
s=1500
0.000056
1500.
expr $f1/$f2
1
X = s / R*sqrt(2*b*p/3) + (t_RTO * (3*sqrt(3*b*p/8) * p * (1+32*p^2)))

X is the transmit rate in bytes/second.

s is the packet size in bytes.

R is the round trip time in seconds.

p is the loss event rate, between 0 and 1.0, of the number of loss events as a fraction of the number of packets transmitted.

t_RTO is the TCP retransmission timeout value in seconds.

b is the number of packets acknowledged by a single TCP acknowledgement.

Figure 3.3: Implementation of the TFRC equation

As recommended in [28], *t_RTO* should be set to either 4×*R* or alternatively max(4×*R*, 1 second), where the Round Trip Time *R* is two times the latency estimate given by **jit.net.send**. The packet size *s* is set to 1500 Bytes, here, which is the Maximum Transmission Unit (MTU) size in an Ethernet LAN [34], while *b* = 1 in [28], While parameters *s* and *b* are fixed, other three parameters *R*, *p*, and *t_RTO* (which is derived from *R*) are dynamic and can change during the streaming session. Jitter object **jit.net.send** provides an estimate of one-way latency (i.e., half the value of *R*), and thereby also enables the computation of *t_RTO*. However, **jit.net.send** does not provide an estimate of the packet loss rate *p*. This parameter needs to be estimated by other means.

In this work, we mainly used TFRC for combined control of bit rate and frame rate in a controlled environment set up by the network emulator. In these experiments, described later in this chapter, the network emulator was set up to provide constrained bandwidth, but didn't introduce any loss. If we plug $p = 0$ into the TFRC equation above, the resulting bandwidth estimate would be $X = \infty$, which is not realistic. Hence, we decided to test various small values of p in the TFRC equation while running the experiment across the network emulator, to see how closely the TFRC estimate would be to the bandwidth constraint set by the emulator. Figures 3.4. and 3.5 show example Jitter patches used in these tests.

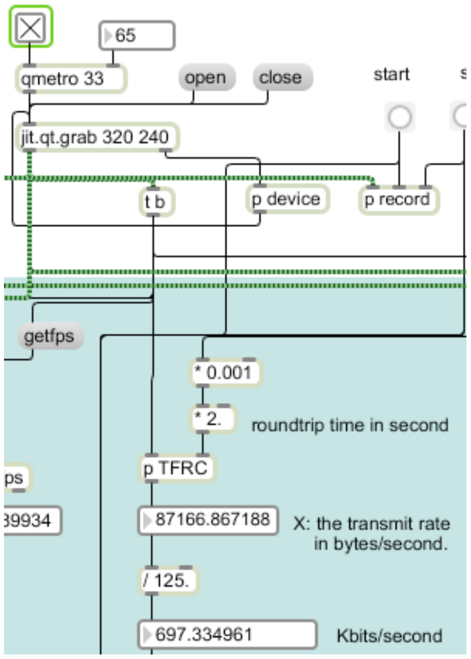


Figure 3.4: TFRC patch calculates the available bandwidth in Kbps, while the network emulator was set to provide 700 Kbps

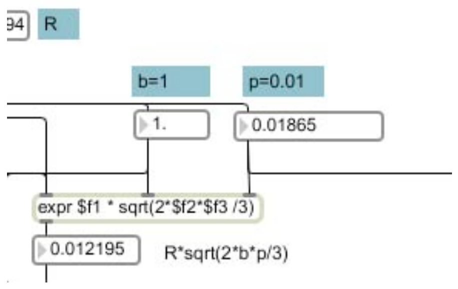


Figure 3.5: Loss event rate *p* in subpatch "p TFRC" at 700 Kbps

Network emulator setting	qmetro	Measured latency (ms)	Manually set *p*	TFRC estimate (Kbps)
700 Kbps	65	54	0.01865	700.0 – 700.4
600 Kbps	75	63	0.01870	599.0 – 600.2
550 Kbps	80	70	0.02030	549.0 – 550.0

Table 3.1: *p* values set to estimate bandwidth of 700 Kbps, 600 Kbps, and 550 Kbps

Table 3.1 shows the values of *p* that worked out best in estimating the bandwidth set by the network emulator, which is shown in the first column. Hence for the remaining experiments, we set the value of *p* to 0.019, which is the average of the values shown in the fourth column of the table. With this value of *p* in the TFRC equation, the estimated bandwidth in the experiments across the network emulator is very close to the true bandwidth constraint.

3.1.3 Encoding bit rate control by TFRC

Now that we are able to estimate the available bandwidth within Max/MSP/Jitter, the next step is to use this bandwidth estimate to control the video encoding bit rate. If *X* is the TFRC bandwidth estimate in bits per second (bps), and *F* is the current frame rate in frames per second (fps), then the number of bits that should be devoted to the next frame is *X* / *F*. If *W* and *H* are the width and height of the frame, then the bit rate in bits per pixel (bpp) is given by *X* / (*F*×*W*×*H*). This is the value that is fed to the input of the SPIHT encoder. A Jitter patch that accomplishes this conversion is shown in Figure 3.6, where the dimensions of the frame are 320×240.

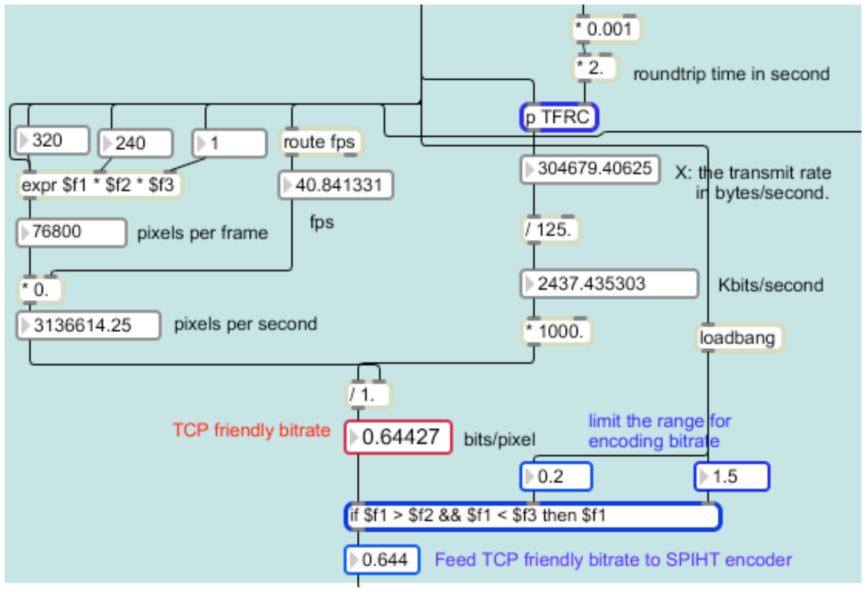


Figure 3.6: A patch to convert TFRC bit rate from bits per second into bits per pixel

As demonstrated in an earlier chapter, the frame quality with 320×240 resolution is already very good when the encoding bit rate is 1.0 bpp, and visually lossless at 1.5 bpp. Hence, in the patch in Figure 3.6, we set the upper limit on the encoding rate to 1.5 bpp, because higher encoding bit rates would not lead to visible improvement in quality, and would simply waste bandwidth. Similarly, we judged that the lowest acceptable visual quality is achieved around 0.2 bpp, so we set this value as the lower limit of the encoding bit rates.

The above procedure provides a control policy for setting the encoding bit rate of each video frame based on the current estimate of the available bandwidth and the current frame rate. The next step is to determine the appropriate frame rate for a given scene. To accomplish this, we need to be able to detect motion and estimate the amount of motion intensity in the scene.

3.2 Motion detection

To detect motion between two frames and estimate the corresponding motion intensity, we use the approach from [37], which is based on histogram of differences (hod). If f_n and f_m denote two frames, then the intensity of motion between them is calculated as

$$D_h(f_n, f_m) = \frac{\sum_{i>|TH_0|} hod(i)}{N_{pixel}}, \tag{4}$$

where *hod* is the histogram of pixel-wise differences between the two frames, *i* is the index of the *hod* bin, TH_0 is the threshold value for the minimum difference considered to be significant, and N_{pixel} is the number of pixels in the frame. The value of D_h represents the percentage of pixels whose values differ significantly between the two frames. This approach is fairly simple, yet it provides a useful measure of motion intensity between f_n and f_m.

We have developed a Jitter external object called **mcl.jit.motion** which implements the above approach. As shown in Figure 3.7, the **mcl.jit.motion** object allows the user to set the threshold value TH_0. In this example, the value of TH_0 is set to 0.125. Note that in the context of our **mcl.jit.motion** object, TH_0 refers to a normalized range of pixel values [0, 1]. Hence, 0.125 corresponds to the actual pixel value difference of $0.125 \times 255 = 31.875$.

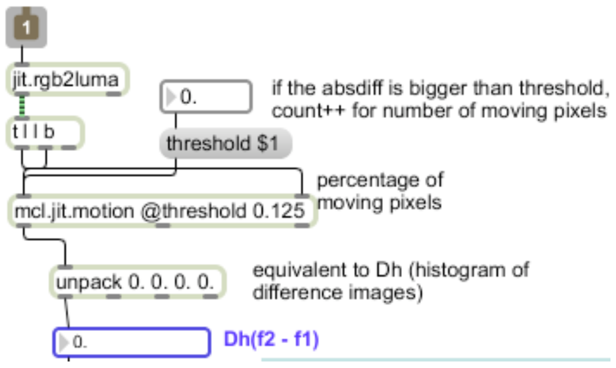
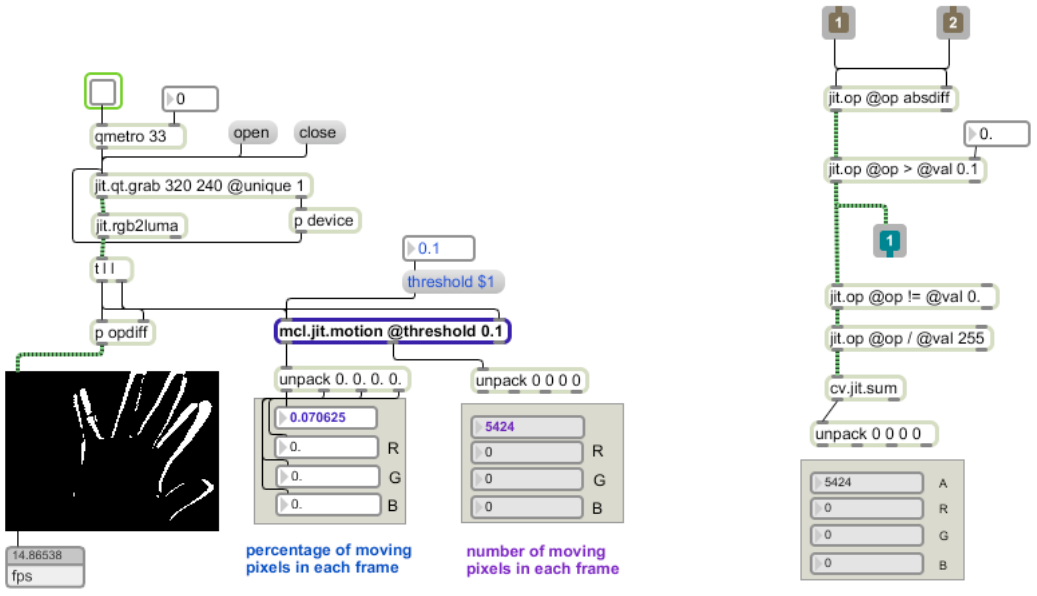

1
jit.rgb2luma
0.
if the absdiff is bigger than threshold, count++ for number of moving pixels
t l l b
threshold $1
percentage of moving pixels
mcl.jit.motion @threshold 0.125
unpack 0. 0. 0. 0.
equivalent to Dh (histogram of difference images)
0.
Dh(f2 - f1)

Figure 3.7: Motion detection object with TH_0 = 0.125

3.2.1 Using motion detection in Max/MSP/Jitter

Video frames are grabbed from the camera into the Max/MSP/Jitter environment using the standard Jitter object **jit.qt.grab**. If the **qmetro** parameter, which determines the frame grabbing interval, is too short for a particular camera, the **jit.qt.grab** object will simply duplicate the last captured frame. This means that it is possible to end up in a situation where neighbouring frames in Max/MSP/Jitter are identical regardless of the motion in the scene. Hence, in order to successfully use the **mcl.jit.motion** object within Max/MSP/Jitter, the **jit.qt.grab** object needs to be followed by an attribute "**@unique 1**," as shown in Figure 3.8(a). This attribute will force the **jit.qt.grab** object to output only distinct frames. With this attribute set, the resulting frame rate will not exceed the maximum possible frame rate of the camera, even if the **qmetro** parameter is set to a very low value.

The **mcl.jit.motion** object can detect motion in both monochrome and colour video. For use with monochrome video, we insert a **jit.rgb2luma** object to convert RGB video frame into a 1-plane char monochrome video frame. As illustrated in Figure 3.8(a), live video matrices from **jit.qt.grab** are fed into “**t l l**” object, where “**l**” stands for “list”. This means that two neighbouring video frames from **jit.qt.grab** are stored in the tab before get passed to the **mcl.jit.motion** object. The sub-patch “**p opdiff**” is used to do the frame subtractions, and its subtracted images are displayed in the jitter window in Figure 3.8(a).



(a) Motion detection patch (b) Sub-patch “p opdiff”

Figure 3.8: Motion detection patch for greyscale video frames

The **mcl.jit.motion** object is designed with two outlets. The first outlet outputs the percentage of pixels with significant motion, that is, the value of D_h in equation (4). The second outlet outputs the actual number of pixels with significant motion, $D_h \times N_{pixel}$.

The sub-patch “**p opdiff**” (Figure 3.8(b)) was used in this example to display the difference between the last two frames, and also to double-check the results from the second outlet of **mcl.jit.motion** object. In this sub-patch, inlets 1 and 2 take the two frames from "**t l l**" and feed them into the "**jit.op@op absdiff**" object for pixel-by-pixel subtraction. The next object, "**jit.op@op > @val 0.1**," compares the result of subtraction with a threshold of 0.1: if the value entering the comparison is larger than the threshold, a value of 255 will be passed to the output, otherwise a 0 value will be passed. Hence, outlet 1 in the sub-patch will contain a matrix of the same resolution as the video frame, and will have a white pixel value (255) in all pixels which seem to be moving, and a black pixel value (0) in all pixels that seem to be static, as displayed in Figure 3.8 (a). The two **jit.op** objects after the "**jit.op@op > @val 0.1**" object are there to normalize the values in the matrix to [0, 1], so that each value of 255 in the matrix will be mapped to 1. Then, the **cv.jit.sum** object sums up all the values in the normalized matrix, which is equivalent to counting the non-zero values. In Fig. 3.8(a), we can see that the number of non-zero (i.e., moving) pixels counted by the sub-patch is the same as the number obtained by the **mcl.jit.motion** object (5424), which verifies the correctness of implementation of motion detection in the **mcl.jit.motion** object.

The same operations can be performed on RGB frames. In this case, we do not need the **jit.rgb2luma** object, and the **mcl.jit.motion** object will output the percentages and counts of moving pixels in RGB planes separately, as shown in Figure 3.9 below.

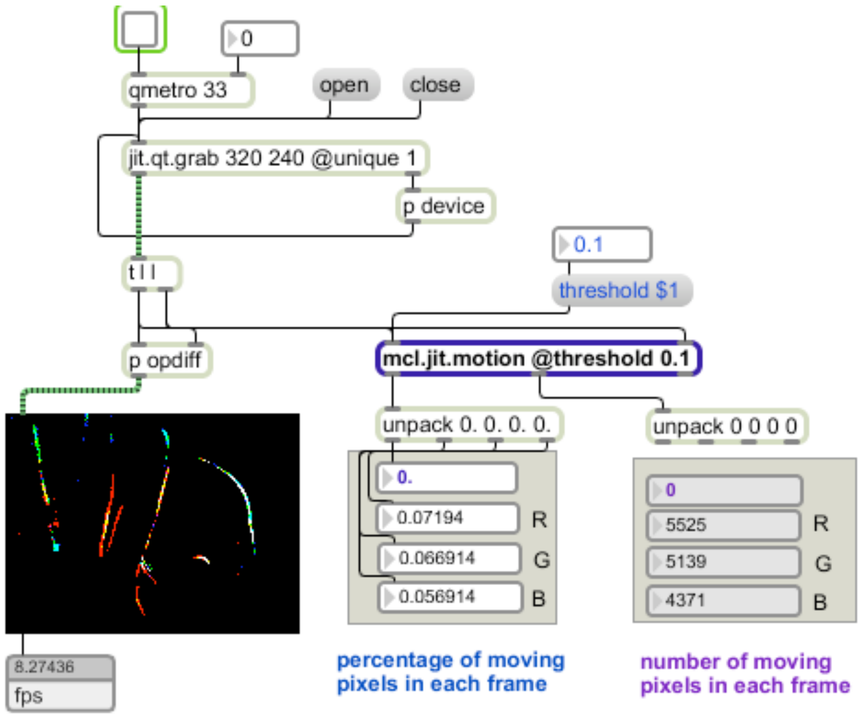


Figure 3.9: Motion detection patch for RGB video frames

3.2.2 Application in dance performances

We have used the **mcl.jit.motion** object in a dance performance called Imprint [3], premiered in the Museum Of Anthropology (MOA) in Vancouver in January 2010. The live video was captured from the surveillance cameras installed at the two sides of a large screen in the Press Centre at MOA. The **mcl.jit.motion** object was used to detect motion in the live video, and obtain outlines of the moving objects, similar to what is shown in Figs. 3.8 and 3.9. These moving object outlines were then overlaid onto the darkened scene captured by the camera, which created a visual effect of moving objects glowing against the darkened background. Such augmented reality scene was displayed on the large screen facing the audience, as shown in Figure 3.10.

Figure 3.10: Augmented reality scene used in the Imprint dance performance at MOA

3.3 Frame rate control based on motion trend

In this section, we briefly describe the frame rate control method proposed in [37]. Since this method was developed by researchers from the University of Southern California, we will refer to it as the "USC method" for convenience. This paper introduced a rate control algorithm with variable frame rate control and bit allocation for H.263+ [40] video coding. They compared their coded videos with the existing rate control algorithms, such as TMN5 [38] and TMN8 [39], and concluded that their method could provide a better trade-off between spatial and temporal quality, by avoiding the abrupt frame skipping in a pre-recorded video. In this work, we utilize SPIHT-based video codecs described in the previous chapter, rather than H.263+. Hence, the bit allocation portion of the method from [37] is not relevant here. Instead, we focus on their frame rate control algorithm, which will be described in more detail below. Another important difference between our work and that in [37] is that we consider live video, while [37] considers pre-recorded video.

3.3.1 Frame rate control in the USC method

The USC method [37] was originally developed for combined frame rate and bit rate control in H.263+ video coding, but its frame rate control strategy can be easily adapted to other coding scenarios. In this method, video is divided into sub-Groups Of Pictures (sub-GOPs), and each sub-GOP consists of 12 frames. It is assumed that the video is captured at a certain frame rate, say 30 fps, and the goal is to drop some of these frames (thereby changing the instantaneous frame rate) without affecting visual quality. Since the factors of number 12 are 1, 2, 3, 4, and 6, it is possible to achieve five frame rates derived from the original one by keeping 1, 2, 3, 4, or 6 out of the original 12 frames in a sub-GOP. In Tables 3.2 and 3.3, we show these frame rates relative to the original one, along with the indices of the retained frames: predominantly even-indexed frames in Table 3.2 and predominantly odd-indexed frames in Table 3.3. The values of the **qmetro** parameter needed to achieve these frame rates in Max/MSP/Jitter, assuming the original frame rate is 30 fps, are also given in the tables.

qmetro	Frame rate relative to original	Retained frame indices
34	1/1	1,2,3,4,5,6,7,8,9,10,11,12
68	1/2	2,4,6,8,10,12
102	1/3	3,6,9,12
136	1/4	4,8,12
204	1/6	6,12
408	1/12	6

Table 3.2: Predominantly even-indexed retained frames

qmetro	Frame rate relative to original	Retained frame indices
34	1/1	1,2,3,4,5,6,7,8,9,10,11,12
68	1/2	1,3,5,7,9,11
102	1/3	1,4,7,10
136	1/4	1,5,9
204	1/6	1,7

Table 3.3: Predominantly odd-indexed retained frames

If the original frame rate is 30 fps, this frame rate adjustment scheme can support frame rates from 30/12 = 2.5 fps to 30 fps. Therefore, in our implementation of this scheme in Max/MSP/Jitter, the **qmetro** was allowed to change among 6 discrete levels: 34 68 102 136 204 408, which correspond to a range of frame rates from 2.5 fps to 30 fps. The patches implementing the USC scheme are shown in Figures 3.11 - 3.13.

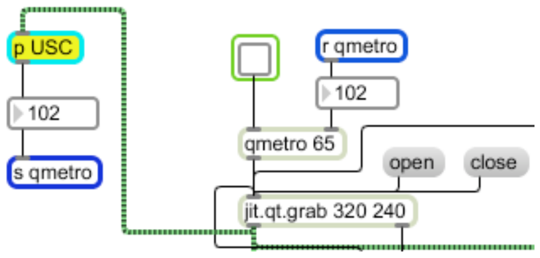


Figure 3.11: The frame rate control patch for the USC method

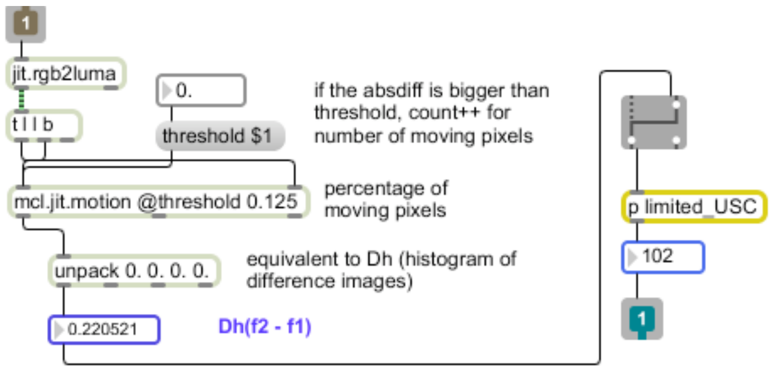

1
jit.rgb2luma
t l l b
0.
threshold $1
if the absdiff is bigger than threshold, count++ for number of moving pixels
mcl.jit.motion @threshold 0.125
percentage of moving pixels
unpack 0. 0. 0. 0.
equivalent to Dh (histogram of difference images)
0.220521
Dh(f2 - f1)
p limited_USC
102
1

Figure 3.12: Calculate the histogram of difference images by mcl.jit.motion

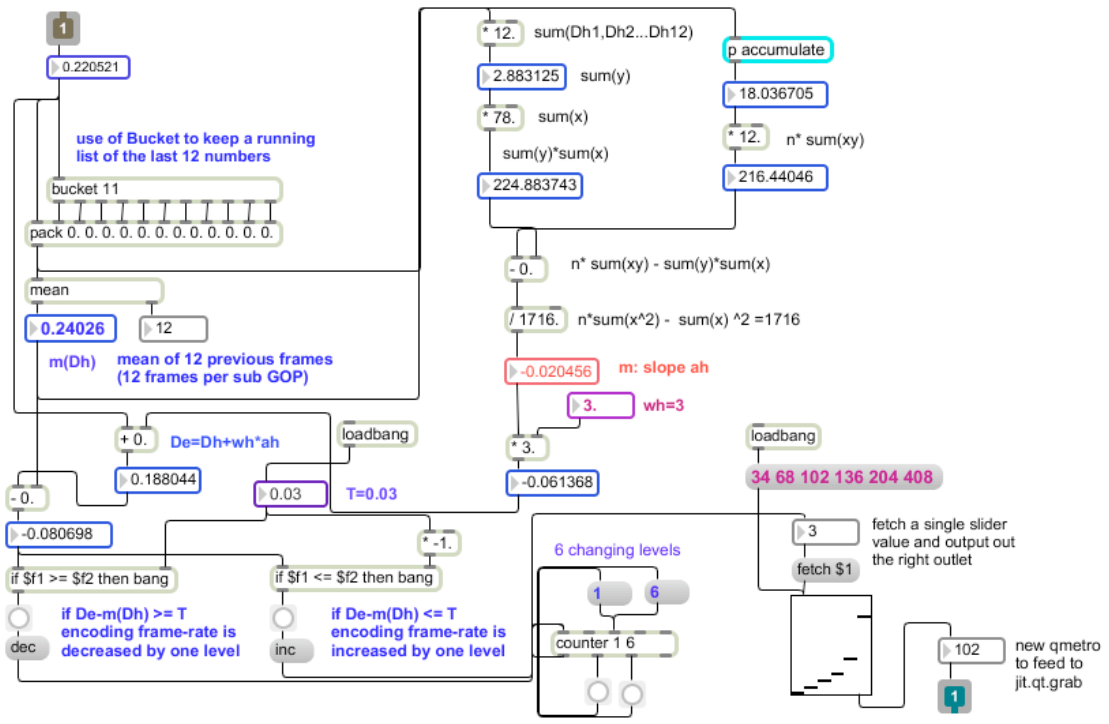

1
0.220521
use of Bucket to keep a running list of the last 12 numbers
bucket 11
pack 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
mean
0.24026
12
m(Dh)
mean of 12 previous frames (12 frames per sub GOP)
+ 0.
De=Dh+wh*ah
0.188044
loadbang
0.03
T=0.03
- 0.
-0.080698
if $f1 >= $f2 then bang
if De-m(Dh) >= T encoding frame-rate is decreased by one level
dec
* -1.
if $f1 <= $f2 then bang
if De-m(Dh) <= T encoding frame-rate is increased by one level
inc
* 12.
sum(Dh1,Dh2...Dh12)
2.883125
sum(y)
* 78.
sum(x)
sum(y)*sum(x)
224.883743
p accumulate
18.036705
* 12.
n* sum(xy)
216.44046
- 0.
n* sum(xy) - sum(y)*sum(x)
/ 1716.
n*sum(x^2) - sum(x) ^2 =1716
-0.020456
m: slope ah
3.
wh=3
* 3.
-0.061368
loadbang
34 68 102 136 204 408
6 changing levels
1
6
counter 1 6
3
fetch $1
fetch a single slider value and output out the right outlet
102
new qmetro to feed to jit.qt.grab
1

Figure 3.13: Sub-patch “p HOD” which specifies 6 discrete levels for qmetro

The USC method operates as follows. The D_h values (eq. (4)) are computed for all 12 frames in the current sub-GOP. Then the D_h value for the next sub-GOP, denoted $\hat{D_e}$, is predicted as follows:

$$\hat{D_e} = D_h + w_h a_h , \tag{5}$$

where D_h in (5) is obtained from the last two encoded frames, a_h is the slope of the linear least squares fit to the previous 12 values of D_h, and w_h is a weight factor set to 3 in [37].

By comparing the predicted value of D_h from eq. (5) with the mean value m(D_h) of the last 12 values of D_h, one can tell whether the motion has an increasing or decreasing trend. Let $\delta = \hat{D_e} - m(D_h)$. The frame rate for the next sub-GOP is adjusted as follows:

1) If $\delta \geq T$, the frame rate is decreased.

2) If $\delta \leq -T$, the frame rate is increased.

3) Otherwise, the frame rate is unchanged.

Here, T is the threshold chosen as the average D_h over the first 12 video frames. T is set to 0.03 at the beginning of the video (i.e., before the first 12 frames) are processed, as indicated in [37]. In addition, to prevent sudden changes in frame rate, the authors in [37] limited the change of frame rate to only one neighbouring level. In other words, if the current frame rate is 1/3 of the original, the next frame rate can only be chosen to be 1/2, 1/3, or 1/4 of the original frame rate.

The main drawback of this method is that it may be too slow in changing the frame rate when the motion suddenly appears in a previously static scene. The estimated motion trend a_h is computed over 12 previous frames. Hence, it may take several frames with significant motion until the trend changes significantly enough to trigger the frame rate increase, which in turn may cause video to look jerky. Therefore, we developed a method that has faster response to a sudden increase in motion, which will be described in Section 3.4.

3.4 Frame rate control based on instantaneous motion

To avoid slow reaction time associated with monitoring the motion trend, we propose to adjust the frame rate based on instantaneous motion. We borrowed the idea of Additive Increase Multiplicative Decrease (AIMD) from TCP flow control [43] for controlling the frame sampling interval τ, which is the time between two captured frames. In our proposed scheme, the new value τ_{new} is chosen based on the D_h value of the last two captured frames as follows:

1) If $D_h > T_d$, then $\tau_{\text{new}} = \tau_{\text{prev}} \times \alpha$.

2) If $D_h \leq T_d$, then $\tau_{\text{new}} = \tau_{\text{prev}} + \beta$.

Here, $0 < \alpha < 1$, $\beta > 0$, and T_d is the threshold which is set as the maximum value of D_h between two neighbouring frames that is not perceived as jerky motion. To enable a rapid increase in frame rate (i.e., rapid decrease in τ) when the motion level increases, we need to set a small value to α. Hence, α is critical for the performance of the method, and the appropriate value for it is determined through extensive subjective testing described later in the chapter. On the other hand, the frame rate does not need to be decreased (i.e., τ does not need to be increased) rapidly when the motion in the scene dies out, since having a frame rate slightly higher than necessary will not produce jerky video. Hence, the value of β is not as crucial. We tested several possible values for β and determined that β = 100 milliseconds works well for this purpose.

To empirically determine the value for T_d, we tested several typical movements that could be expected in videoconferencing (e.g., talking head, hand wave, etc.) and observed the values of D_h. When the scene was static, the D_h value output from **mcl.jit.motion** was ranging from 0 to 0.002, because camera sensor noise can lead to pixel value difference between neighbouring frames even without motion in the scene. When there was a moving object in front of the camera, the D_h value would typically increase above 0.008. Experimenting with different values of T_d in the procedure above, we determined that T_d = 0.01 works fairly well, and this is the value we used in the remaining experiments.

3.4.1 Implementation in Max/MSP/Jitter

The frame sampling interval τ is equal to the **qmetro** parameter value in Max/MSP/Jitter. Hence, the above control procedure is applied to the **qmetro** value in the patches shown in Figures 3.14 and 3.15. To make the range of frame rates of our method be compatible with the USC method, we limited the range of **qmetro** values to [34, 408]. However, in our method, **qmetro** is not constrained to only 6 discrete values - any integer value in this range is allowed.

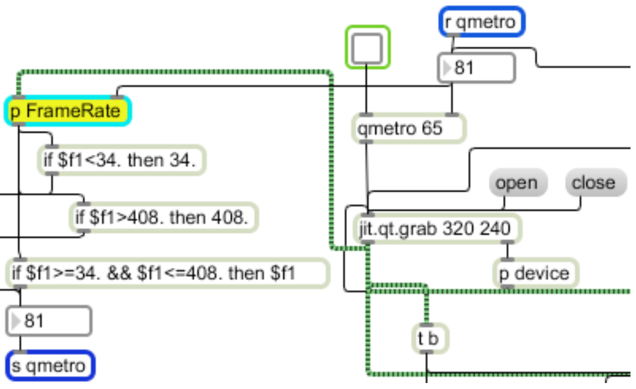
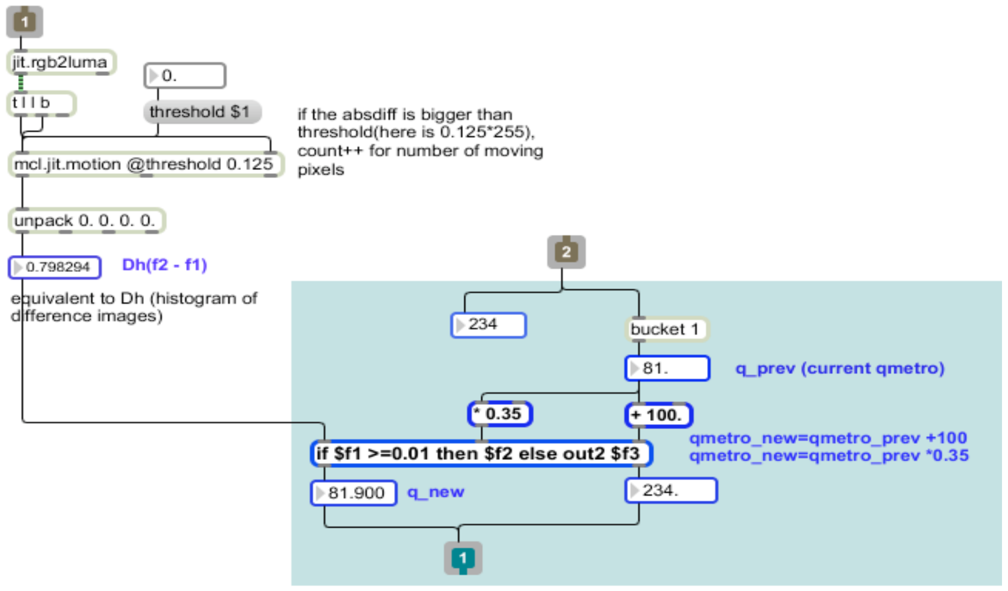


Figure 3.14: Compute new qmetro (value between 34 and 408) for jit.qt.grab



1
jit.rgb2luma
0.
t l l b
threshold $1
if the absdiff is bigger than threshold(here is 0.125*255), count++ for number of moving pixels
mcl.jit.motion @threshold 0.125
unpack 0. 0. 0. 0.
2
0.798294
Dh(f2 - f1)
equivalent to Dh (histogram of difference images)
234
bucket 1
81.
q_prev (current qmetro)
* 0.35
+ 100.
qmetro_new=qmetro_prev +100
qmetro_new=qmetro_prev *0.35
if $f1 >=0.01 then $f2 else out2 $f3
81.900
q_new
234.
1

Figure 3.15: Sub-patch "p FrameRate" with our method

3.5 Combined frame rate and bit rate control

With the appropriate frame rate *F* determined by one of the two methods (USC or ours), and the available bandwidth *X* estimated through TFRC, the combined control of frame rate and bit rate is achieved by setting the current frame rate to *F* and the number of bits assigned to the next frame as *X* / *F*. Combined control patch is shown in Figure 3.16.

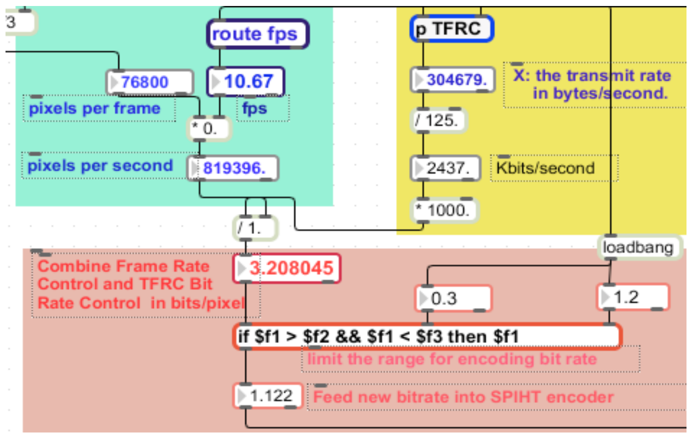

3
route fps
76800
10.67
pixels per frame
fps
* 0.
pixels per second
819396.
/ 1.
p TFRC
304679.
X: the transmit rate in bytes/second.
/ 125.
2437.
Kbits/second
* 1000.
loadbang
Combine Frame Rate Control and TFRC Bit Rate Control in bits/pixel
3.208045
0.3
1.2
if $f1 > $f2 && $f1 < $f3 then $f1
limit the range for encoding bit rate
1.122
Feed new bitrate into SPIHT encoder

Figure 3.16: Combined frame rate and bit rate control

3.6 Comparison of frame rate control methods

In order to compare the two frame rate control methods (USC and ours), we designed the test bed shown in Figure 3.17. The goal of this comparison is to determine which frame rate control method (USC or ours), when coupled with TFRC for bit rate control, produces better trade off between frame quality and frame rate for live video. Since there are currently no widely accepted quantitative metrics for evaluating this trade off, we resort to the ultimate test - subjective evaluation.

The setup for the subjective evaluation includes four computers connected to the Simena NE3000 Network Emulator. Two computers (A and C) perform frame rate control and encoding on the same input video and send the resulting bit streams to the receivers (B and D) through the network emulator, which limited both streams (A → B and C → D) to the same bandwidth in bits/second. A fifth computer (not shown in the figure) is connected to the main port of the emulator to control the bandwidth.

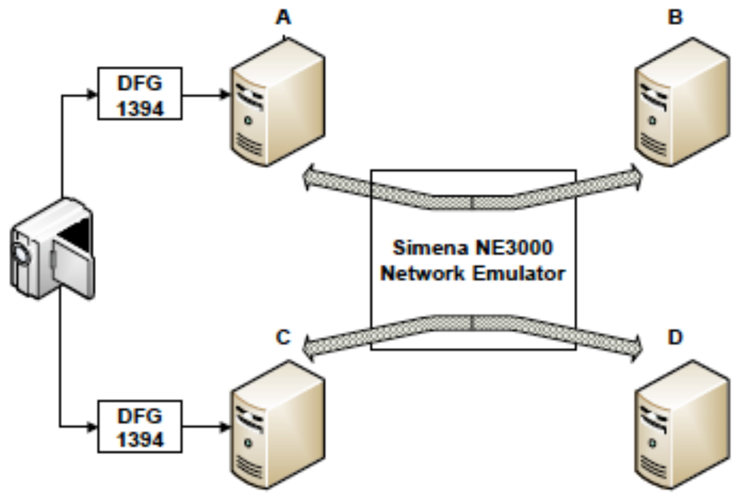
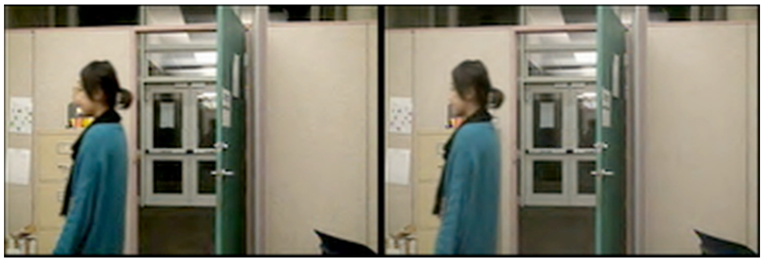


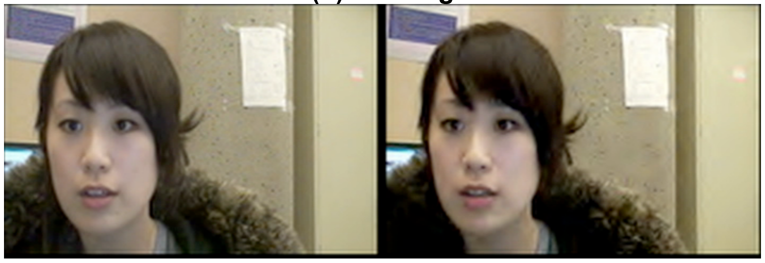
Figure 3.17: Experimental test bed

Two rate control patches, one based on USC frame rate control, the other based on our method, were running on two PC machines A and C. The live analogue video from a Sony HDR-SR12 camera was split into two signals by a RCA splitter cable, and fed into two Imaging Source DFG 1394 video converters. Two MacBook Pro laptops were used as receivers B and D. They also recorded the decoded video in real time into an uncompressed QuickTime format at 60 fps. This is because the DFG converters were limited to 30 fps, so a higher frame rate of 60 fps for storing the received videos ensured that no received frame would be skipped. Using this setup, we recorded four movements that may be considered typical in mobile videoconferencing: *Camera pan*, *Walking, Talking head*, and *Handwave*. Sample frames from the corresponding video clips (one from the USC method, the other from ours) are shown side by side in Figure 3.18. Recordings were made for three values of α (0.25, 0.35, and 0.5), each using four bandwidth settings (400, 850, 1000, and 1500 Kbps). Hence, there were a total of $3 \times 4 \times 4 = 48$ video recordings (three values of α, four bandwidths, and four movement types).

The recorded videos were compared using extensive subjective evaluations with 22 non-expert participants (6 women and 16 men) in a classroom at Simon Fraser University. All participants had normal or corrected-to normal visions. The Two Alternative Forced Choice (2AFC) method [61] was used for comparing the video quality produced by the two frame rate control methods. In 2AFC, participants watch two videos of the same scene produced by the two frame rate controlled methods under the same conditions, and select one of the videos they like more. In other words, there are two alternatives, and the participant is forced to make a choice. If the two clips look equally good, the participants would make a random choice between the two clips, which is expected to cancel out across the participants, so in that case both methods would get approximately the same number of votes. The order of playing video clips was randomized. In addition, each clip was played to each participant twice: once on the left side, once on the right. This was done to eliminate any potential bias that the participant may have for the left or right side.

(a) Camera pan


(b) Walking


(c) Talking head

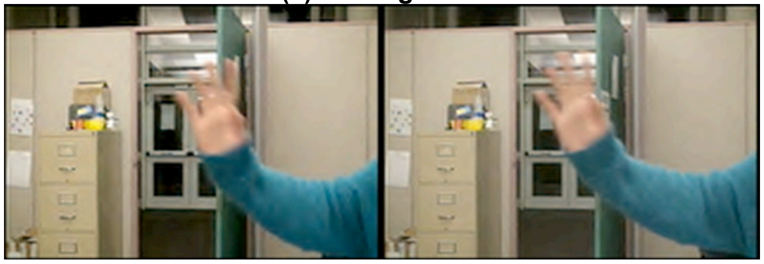

(d) Handwave

Figure 3.18: Four motion types in our experiments

These subjective tests were performed one participant at a time over the period of two days in June 2010. We used a MacBook Pro (2.5 GHz Intel Core 2 Duo) computer playing the 96 clips on a Samsung SyncMaster 915N monitor. Each participant was seated at a distance of 80 cm from the monitor, and was shown pre-recorded instructions on how to complete the evaluation before the test. Details of the test conditions are shown in Table 3.4.

Ambient light	197 Lux
Monitor type	Samsung SyncMaster 915N
Size of video on the screen	332mm × 118mm (width × height)
Distance of viewer from the screen	80 cm
Participants	22 (16 men, 6 women)

Table 3.4: Test conditions

3.7 Results of Subjective Evaluations

Each pair of video clips was shown to each participant twice, the total number of votes under each set of test conditions (movement, α, bit rate) is 2×22 = 44. The results for the three values of α are summarized in Tables 3.5-3.7. A result is called *statistically significant* if it is unlikely to have occurred by chance. We used chi-square (χ^2) test [61] to examine the statistical significance of each result:

$$\chi^2 = \sum_{j=1}^{n} \frac{(O_j - E_j)^2}{E_j} \tag{8}$$

where O_j is the observed count of the j-th outcome, E_j is the expected count of the j-th outcome asserted by the null hypothesis, and n is the number of possible outcomes. Since we are comparing two frame rate control methods (USC and our own), n = 2 in our experiment. The χ^2 value can be used to compute the statistical significance value (also known as p-value), which are also shown in the tables. As a rule of thumb in experimental sciences, results with $p < 0.05$ are considered statistically significant.

3.7.1 Frame rate control comparison by α values

Votes for α =0.25 are shown in Table 3.5.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	23	21	0.7630	
	850	17	27	0.1317	
	1000	8	36	<0.0001	Yes
	1500	14	30	0.0159	Yes
Camera pan	400	18	26	0.2278	
	850	6	38	<0.0001	Yes
	1000	14	30	0.0159	Yes
	1500	21	23	0.7630	
Walking	400	22	22	1.0000	
	850	11	33	0.0009	Yes
	1000	22	22	1.0000	
	1500	15	29	0.0348	Yes
Handwave	400	16	28	0.0704	
	850	9	35	<0.0001	Yes
	1000	7	37	<0.0001	Yes
	1500	3	41	<0.0001	Yes
Total votes		226	478	<0.0001	Yes

Table 3.5: Votes for α = 0.25

As seen in the table, all statistically significant results (9 out of 16) show preference for our scheme. In addition, preference for our scheme seems to be stronger at higher bit rates.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	21	23	0.7630	
	850	12	32	0.0026	Yes
	1000	8	36	<0.0001	Yes
	1500	12	32	0.0026	Yes
Camera pan	400	11	33	0.0009	Yes
	850	15	29	0.0348	Yes
	1000	9	35	<0.0001	Yes
	1500	11	33	0.0009	Yes
Walking	400	21	23	0.7630	
	850	15	29	0.0348	Yes
	1000	16	28	0.0704	
	1500	9	35	<0.0001	Yes
Handwave	400	4	40	<0.0001	Yes
	850	6	38	<0.0001	Yes
	1000	27	17	0.1317	
	1500	6	38	<0.0001	Yes
Total votes		203	501	<0.0001	Yes

Table 3.6: Votes for α = 0.35

Votes for α =0.35 are shown in Table 3.6. All statistically significant results (12 out of 16) show preference for our scheme. In this case, preference seems to hold across different bit rates. Finally, votes for α =0.5 are shown in Table 3.7. Again, all statistically significant results (10 out of 16) show preference for our scheme.

Looking across tables for all three α values, there is a clear preference for our method over the USC frame rate control scheme. The number of statistically significant trials is the highest at α = 0.35 (12 out of 16 as shown in Table 3.6), which suggests that this α value is best suited for our method.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	17	27	0.1317	
	850	8	36	<0.0001	Yes
	1000	14	30	0.0159	Yes
	1500	8	36	<0.0001	Yes
Camera pan	400	6	38	<0.0001	Yes
	850	10	34	0.0003	Yes
	1000	18	26	0.2278	
	1500	18	26	0.2278	
Walking	400	21	23	0.7630	
	850	18	26	0.2278	
	1000	13	31	0.0067	Yes
	1500	11	33	0.0009	Yes
Handwave	400	6	38	<0.0001	Yes
	850	13	31	0.0067	Yes
	1000	9	35	<0.0001	Yes
	1500	16	28	0.0704	
Total votes		206	498	<0.0001	Yes

Table 3.7: Votes for α = 0.5

When α = 0.25, our method is very aggressive in increasing the frame rate when motion is suddenly detected, and captures more frames than the USC method. On the other hand, the quality of individual frames will be poorer compared with the USC method, since the total bit rate is fixed, so frames in our method have fewer bits assigned to them on average. Therefore, when α = 0.25, the statistically significant preference for our method tends to be at higher bit rates, where the penalty for having a high frame rate does not degrade the individual frame quality too much.

As α value is increased towards 0.5, our method becomes less aggressive in increasing the frame rate. Hence, there will be fewer frames produced and each frame will be assigned more bits, which results in a higher frame quality even if the bit rate is low. As α increases, the statistically significant preference for our method shifts to lower bit rates. This is especially evident for *Handwave* and *Camera pan* types of motion, which had relatively higher motion intensity compared *Talking head* and *Walking*.

A plot of instantaneous frame rate versus time for a segment from the *Camera pan* type of motion encoded at 400 kbps with α =0.35 is shown in Figure 3.19. Both our method and USC method covered a range of frame rates from about 3 fps up to almost 30 fps over this segment of time. The USC method was slower to react to sudden increases in motion intensity, as it relies on motion trend of 12 previous frames to adjust the frame rate. By contrast, our method increases the frame rate right after the first high-motion frame.

As shown in Figure 3.19, by the time the USC method has increased the frame rate to 30 fps, our method has already been operating at that frame rate for a few hundred milliseconds, and has therefore captured more frames in the initial portions of the high-motion segments. This made the motion in the captured video less jerky compared to the USC method. Motion jerkiness is not the same as frame rate fluctuation. In fact, the frame rate produced by our method shows a lot more fluctuation than the frame rate produced by the USC method, but the participants still showed statistically significant preference for our method. This means that frame rate fluctuation is not detrimental to subjective video quality as long as the frame rate is kept above the level that is needed to represent the motion in the scene properly. This finding seems to be in line with [60], but contradicts the claims made in [37] and [41] that human visual system is sensitive to large sudden changes in frame rate. The crucial factor seems to be whether the frame rate is high enough, not whether it fluctuates too much.

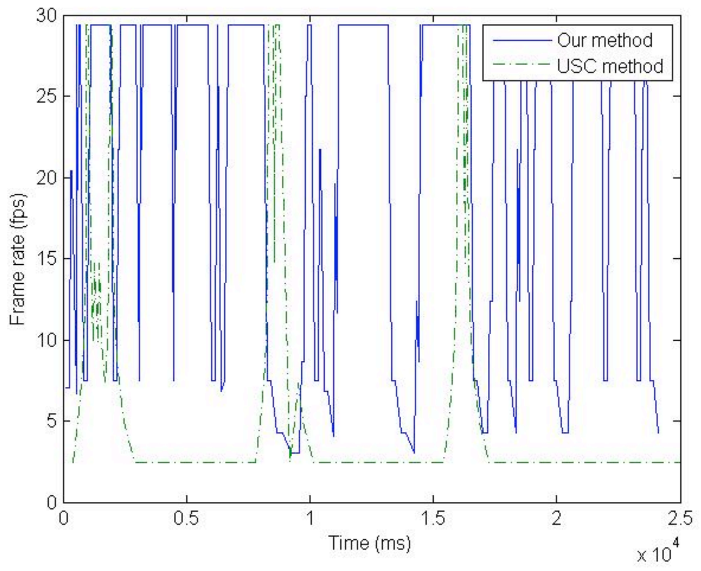


Figure 3.19: Frame rate vs. time for *Camera pan* @ 400kbps with α =0.35

3.7.2 Frame rate control comparison by viewers' prior experience

In this section, we compare the survey results based on viewers' prior experience with videoconferencing. Out of 22 participants, 10 of them said that they used videoconferencing often (no less than once per month), while the other 12 used it occasionally (less than once per month). It is interesting to compare the preference of those who do videoconferencing often (and hence have significant prior experience with it), and those who do it only occasionally.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	11	9	0.6547	
	850	9	11	0.6547	
	1000	4	16	0.0073	Yes
	1500	8	12	0.3711	
Camera pan	400	8	12	0.3711	
	850	4	16	0.0073	Yes
	1000	8	12	0.3711	
	1500	9	11	0.6547	
Walking	400	10	10	1.0000	
	850	3	17	0.0017	Yes
	1000	4	16	0.0073	Yes
	1500	6	14	0.0736	
Handwave	400	5	15	0.0253	Yes
	850	5	15	0.0253	Yes
	1000	3	17	0.0017	Yes
	1500	3	17	0.0017	Yes
Total votes		100	220	<0.0001	Yes

Table 3.8: Votes for α = 0.25 among participants who do videoconferencing often

For α =0.25, votes of viewers who do videoconferencing often (10 people) are shown in Table 3.8. All statistically significant results (8 out of 16) show preference for our scheme. For the same α = 0.25, votes of viewers who do videoconferencing occasionally (12 people) are shown in Table 3.9. Again, all statistically significant results (8 out of 16) show preference for our scheme.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	12	12	1.0000	
	850	8	16	0.1025	
	1000	4	20	0.0011	Yes
	1500	6	18	0.0143	Yes
Camera pan	400	10	14	0.4142	
	850	2	22	<0.0001	Yes
	1000	6	18	0.0143	Yes
	1500	12	12	1.0000	
Walking	400	12	12	1.0000	
	850	8	16	0.1025	
	1000	2	22	<0.0001	Yes
	1500	9	15	0.2207	
Handwave	400	11	13	0.6831	
	850	4	20	0.0011	Yes
	1000	4	20	0.0011	Yes
	1500	0	24	<0.0001	Yes
Total votes		110	274	<0.0001	Yes

Table 3.9: Votes for α = 0.25 among participants who do videoconferencing occasionally

For α =0.35, votes of viewers who do videoconferencing often are shown in Table 3.10, and again all statistically significant results (5 out of 16) show preference for our scheme.

Movement	**Bit rate (kbps)**	**USC**	**Ours**	***p*-value**	**Significant**
Talking head	400	10	10	1.0000	
	850	6	14	0.0736	
	1000	3	17	0.0017	Yes
	1500	3	17	0.0017	Yes
Camera pan	400	7	13	0.1797	
	850	7	13	0.1797	
	1000	7	13	0.1797	
	1500	8	12	0.3711	
Walking	400	8	12	0.3711	
	850	7	13	0.1797	
	1000	9	11	0.6547	
	1500	7	13	0.1797	
Handwave	400	3	17	0.0017	Yes
	850	3	17	0.0017	Yes
	1000	11	9	0.6547	
	1500	4	16	0.0073	Yes
Total votes		103	217	<0.0001	Yes

Table 3.10: Votes for α = 0.35 among participants who do videoconferencing often

For α =0.35, votes of viewers who do videoconferencing occasionally (12 people) are shown in Table 3.11. Again, all statistically significant results (10 out of 16) show preference for our scheme.

Movement	**Bit rate (kbps)**	**USC**	**Ours**	***p*-value**	**Significant**
Talking head	400	11	13	0.6831	
	850	6	18	0.0143	Yes
	1000	5	19	0.0043	Yes
	1500	9	15	0.2207	
Camera pan	400	4	20	0.0011	Yes
	850	8	16	0.1025	
	1000	2	22	<0.0001	Yes
	1500	7	17	0.0412	Yes
Walking	400	13	11	0. 6831	
	850	8	16	0.1025	
	1000	7	17	0.0412	Yes
	1500	2	22	<0.0001	Yes
Handwave	400	1	23	<0.0001	Yes
	850	3	21	0.0002	Yes
	1000	16	8	0.1025	
	1500	2	22	<0.0001	Yes
Total votes		104	280	<0.0001	Yes

Table 3.11: Votes for α = 0.35 among participants who do videoconferencing occasionally

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	7	13	0.1797	
	850	3	17	0.0017	Yes
	1000	8	12	0.3711	
	1500	5	15	0.0253	Yes
Camera pan	400	5	15	0.0253	Yes
	850	6	14	0.0736	
	1000	10	10	1.0000	
	1500	10	10	1.0000	
Walking	400	11	9	0.6547	
	850	8	12	0.3711	
	1000	5	15	0.0067	Yes
	1500	5	15	0.0253	Yes
Handwave	400	2	18	0.0003	Yes
	850	5	15	0.0253	Yes
	1000	6	14	0.0736	
	1500	8	12	0.3711	
Total votes		104	216	<0.0001	Yes

Table 3.12: Votes for α = 0.5 among participants who do videoconferencing often

For α =0.5, votes of viewers who do video chat often (10 people) are shown in Table 3.12. All statistically significant results (7 out of 16) show preference for our scheme.

Movement	Bit rate (kbps)	USC	Ours	*p*-value	Significant
Talking head	400	10	14	0.4142	
	850	5	19	0.0043	Yes
	1000	6	18	0.0143	Yes
	1500	3	21	0.0002	Yes
Camera pan	400	1	23	<0.0001	Yes
	850	4	20	0.0011	Yes
	1000	8	16	0.1025	
	1500	8	16	0.1025	
Walking	400	10	14	0.4142	
	850	10	14	0.4142	
	1000	8	16	0.1025	
	1500	6	18	0.0143	Yes
Handwave	400	4	20	0.0011	Yes
	850	8	16	0.1025	
	1000	3	21	0.0002	Yes
	1500	8	16	0.1025	
Total votes		102	282	<0.0001	Yes

Table 3.13: Votes for α = 0.5 among participants who do videoconferencing occasionally

For α =0.5, votes of viewers who do video chat occasionally (12 people) are shown in Table 3.13. All statistically significant results (8 out of 16) show preference for our scheme.

As observed in the above tables, both groups of participants (those doing videoconferencing often, as well as those who do it occasionally) showed significant preference for videos produced by our method. With α =0.25 (Tables 3.8-3.9) and with α = 0.5 (Tables 3.12-3.13), both groups of viewers had approximately the same number of significant results, which means that they felt approximately the same about motion smoothness and video quality obtained with these α values.

However, with $\alpha = 0.35$ (Tables 3.10-3.11), which seems to be the most appropriate according to the results from the previous section, the two groups showed different levels of preference. With this α value, viewers doing videoconferencing occasionally showed significant preference for our method in 10 out of 16 trials, while those who do videoconferencing often preferred our method in just 5 out of 16 trials. It is natural to ask why those who do not have much experience with videoconferencing are more likely to choose our method compared to those who have more experience with it? We believe the explanation lies in conditioning. Common consumer-grade videoconferencing systems are limited in bandwidth and processing power, and often suffer from low video frame rates which lead to jerky motion. Those who use videoconferencing often are likely to get used to this phenomenon and don't seem to mind it too much. On the other hand, those with less experience with videoconferencing are presumably more used to TV-quality video with higher frame rates. Hence, they seem to be more sensitive to jerky motion, and tend to prefer videos produced by our method more often than experienced videoconferencing users.

4: CONCLUSIONS AND FUTURE GOALS

In this thesis, we presented the **mcl.jit** library of Max/MSP/Jitter external objects, which we developed for scalable video coding and transmission. We have also tested and demonstrated the performance of these tools in several distributed dance performances conducted by the dancers and media artists from the SFU School for the Contemporary Arts. These tools have enabled telematic dance performance to take place even in venues not equipped with high bandwidth Internet access.

We have also introduced a combined control method to adjust frame rate and encoding bit rate for live video streaming in Max/MSP/Jitter. The bit rate is controlled based on TCP-Friendly Rate Control (TFRC) proposed in [28]. The frame rate is controlled according to the motion intensity detected by our **mcl.jit.motion** object. The Jitter patch for combined rate control could be used in future dance performances to enable a more efficient use of the available bandwidth.

A subjective video quality assessment with 22 participants was carried out to compare the video quality produced by our frame rate control method versus the USC method [37]. The survey results showed a statistically significant preference for our method.

4.1 Future goals

4.1.1 Video coding

In the current **mcl.jit** library, all video coding objects are based on the SPIHT algorithm because of its simplicity, speed, and scalability. Further, all codecs are intra-frame codecs, which means that they do not employ motion compensation. Hence, they do not achieve as much compression as would be possible with more complex motion-compensated codecs. As a future improvement possibility, our next goal is to develop Jitter external objects based on optimized motion-compensated codecs such as X.264 [63], which is a fast implementation of H.264 [65]. This would enable much lower bit rates, albeit at the cost of some added complexity.

4.1.2 Audio coding

We have been focused on live video streaming in the performances [1-4] and have employed audio streaming in Active Space only in Imprint II [4]. In that performance, audio transmission at both ends had to be synchronized precisely, as dancers were using audio to guide their movements at various points in the performance. In this case, all computers were interconnected through a local gigabit network, so we had enough bandwidth to stream uncompressed audio using the Active Space objects **NetMatrix_Send** and **NetMatrix_Recv**. However, if we wish to stream both video and audio through a network with lower bandwidth, we need an audio compression object with high quality and low latency.

Standard **jit.net.send/recv** objects can work with uncompressed audio. On the other hand, **jit.broadcast** can send compressed video streams, but it doesn't support live audio compression. Instead, **jit.broadcast** can stream compressed audio from disk. There are a few third party objects could be used to transmit compressed or uncompressed audio over the network described below.

As far as we know, the most frequently used external objects to stream live audio with compression are “**shoutcast~**” [66] and “**oggcast~**” [67]. Unfortunately, their performance is somewhat unstable, and the latency could be a big problem if we need to synchronize the background music accurately between two distant locations. Another pair of commonly used external objects for audio streaming is “**netsend~**” and “**netrecv~**” [68], which were developed for transmitting uncompressed live audio with low latency. We plan to study the source code of “**netsend~**” and “**netrecv~**” and try to develop our own **mcl.jit** external objects for real-time compression and streaming of audio based on these two existing objects.

4.1.3 Multiple ROI coding

At the moment, our **mcl.jit.spihtROIaritenc/dec** objects support only one ROI. If we use **cv.jit.faces** to detect ROI, then only one face can be encoded as ROI, and other faces in the scene will be treated as background (Figure 4.1). In this case, only the first set of coordinates produced by **cv.jit.faces** is treated as ROI, and others are ignored. Our next goal is to add multiple ROI capability to the existing **mcl.jit.spihtROIaritenc/dec** objects. This means that the coordinates of each ROI, as well as its up-shift factor, will have to be stored in the bit stream header in order to allow the decoder to perform reverse operations.

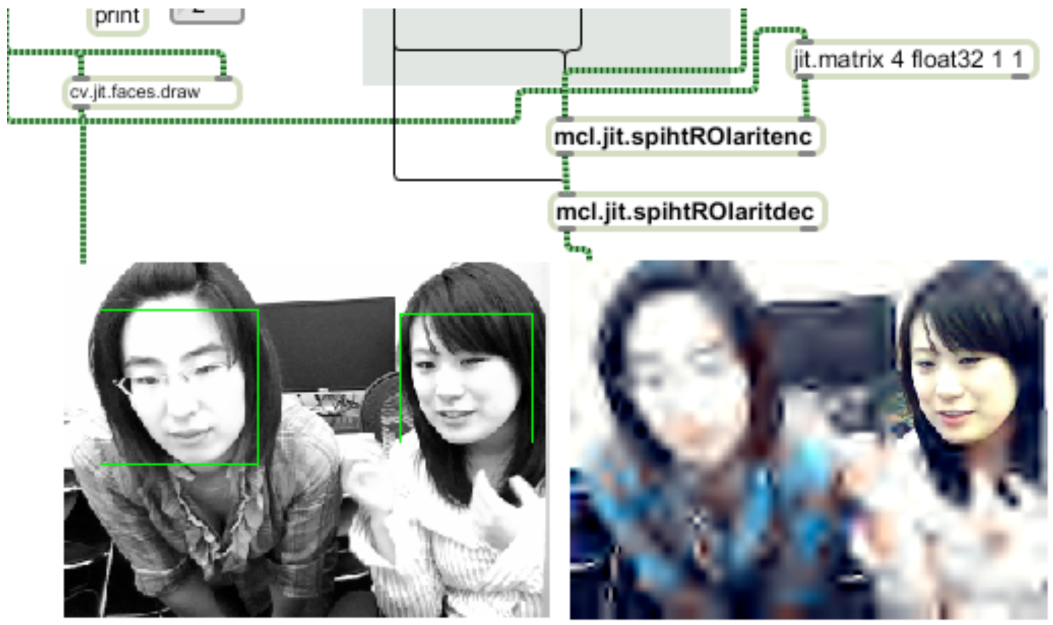


Figure 4.1: Multiple faces are detected by cv.jit.faces, however only the first detected face is treated as ROI by our current ROI codec

REFERENCE LIST

[1] *T2*. Choreography Henry Daniel, Music Adam Basanta & James O'callaghan. Premiered July 11, 2009. Scotia Bank Dance Centre & Interurban Gallery, Vancouver, BC. http://www.sfu.ca/~hdaniel/index.html

[2] *T2: echo.* Choreography Henry Daniel, Music Adam Basanta & James O'callaghan. Premiered November 21, 2009. Concourse Gallery & IDS MoCap Studio, Emily Carr University of Art + Design (ECUAD), Vancouver, BC. http://www.sfu.ca/~hdaniel/index.html

[3] *Imprint.* Choreography Henry Daniel, Music Owen Underhill. Premiered January 23, 2010. Great Hall, Presentation Circle, Bill Reid Rotunda, & Multiversity Galleries, University of British Columbia's Museum of Anthropology, Vancouver, BC. http://www.sfu.ca/~hdaniel/index.html

[4] *Imprint II*. Choreography Henry Daniel, Music Owen Underhill. Premiered June 17, 2010. Fei & Milton Wong Experimental Theatre & the Audain Gallery, SFU Woodward's building, Vancouver, BC. http://www.sfu.ca/~hdaniel/index.html

[5] J. Crawford, "Urban Fabric: Beijing, an interactive dance/media performance" Beijing Dance Academy, 2006.

[6] D. Hall and S. J. Fifer, *Illuminating Video: An Essential Guide to Video Art*, Aperture/BAVC, 1990.

[7] L. Naugle, "Digital dancing," *IEEE Multimedia*, vol. 5, no. 6, pp. 8-12, Oct.-Dec. 1998.

[8] L. Naugle, "Distributed choreography: A video-conferencing environment," *Journal of Performance and Art*, vol. 24, no. 2, pp. 56-62, May 2002.

[9] D. Giuli, F. Pirri, and P. Bussotti, "Orchestra!: A distributed platform for virtual musical groups and music distance learning over the Internet in JavaTM technology," *Proc. IEEE Conf. Multimedia Computing and Systems*, vol. 2, pp. 987-988, Florence, Italy, Jun. 1999.

[10] D. Konstantas, Y. Orlarey, O. Carbonnel, and S. Gibbs, "The distributed musical rehearsal environment," *IEEE Multimedia*, vol. 6, no. 3, pp. 54-64, Jul.-Sep. 1999.

[11] Troika Tronix, *Isadora*, [Online] Available: http://www.troikatronix.com/isadora.html

[12] Cycling'74, *Max/MSP/Jitter*, [Online] Available: www.cycling74.com/products/max5

[13] J.Crawford. "Active Space: Embodied Media in Performance." ACM SIGGRAPH 2005 Sketches. Los Angeles, CA: ACM, 2005. 111. 18 Mar 2009

[14] S. Smulovitz, *Kenaxis*, http://www.kenaxis.com/

[15] J. Burg, *The science of digital media*, Prentice Hall, 2009.

[16] A. Said and W. A. Pearlman, "A new fast and efficient image codec based upon set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–250, Jun. 1996.

[17] Image Compression with Set Partitioning in Hierarchical Trees [Online] http://www.cipr.rpi.edu/research/SPIHT/

[18] I. H. Witten. R. M. Neal. and J. G. Cleary. "Arithmetic coding for data compression," *Commun. ACM*, vol.30. pp.520-540, June 1987

[19] A. Said and W. A. Pearlman, "Reversible Image compression via multiresolution representation and predictive coding," *Proc. SPIE Conf. Visual Communications and Image Processing '93*, Proc. SPIE 2094, pp. 664-674, Cambridge, MA, Nov. 1993

[20] A. Said and W. A. Pearlman, "An Image Multiresolution Representation for Lossless and Lossy Image Compression," *IEEE Transactions on Image Processing,* vol. 5, pp. 1303-1310, Sept. 1996.

[21] I. V. Bajić and X. Ma, "MCL.JIT library for scalable live video in Max/MSP/Jitter," *Proc. IEEE CCECE'10*, Calgary, AB, May 2010.

[22] I. V. Bajić and X. Ma, "Scalable video coding for telepresence in the performing arts," *IEEE ComSoc MMTC E-Letter*, vol. 4, no. 8, pp. 28-30, Sep. 2009.

[23] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*, Kluwer, Norwell, MA, 2001.

[24] M. Rabbani. and P.W.Jones, *Digital Image Compression Techniques*, SPIE Opt. Eng. Press, Bellingham, Washington, 1991.

[25] J. M. Pelletier, CV.JIT: Computer vision for Jitter. [Online] Available: http://www.iamas.ac.jp/~jovan02/cv/

[26] P. Viola and M. J. Jones, "Robust real-time face detection," *Int. J. Comp. Vision*, vol. 57, no. 2, pp. 137-154, May 2004.

[27] J. Goldberg, "T2: Technological beauty but no cohesion," *Plank Magazine*, July 2009.

[28] S. Floyd, M. Handley, J. Pahdye, and J. Widmer, *TCP-Friendly Rate Control (TFRC): Protocol Specification*, RFC 5348, Sep. 2008

[29] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proc.ACM SIGCOMM 2000*, 2000.

[30] M. Miyabayashi, N. Wakamiya, M. Murata, and H. Miyahara. MPEG-TFRCP: Video Transfer with TCP-friendly Rate Control Protocol. In *Proc. IEEE International Conference on Communications (ICC2001)*, pages 137–141, June 2001.

[31] F. Licandro and G. Schembra. A Rate/Quality Controlled MPEG Video Transmission System in a TCP-Friendly Internet Scenario. In *Proc. Packet Video 2002*, 2002.

[32] N. Wakamiya, M. Miyabayashi, Masayuki Murata, and Hideo Miyahara. MPEG-4 Video Transfer with TCP-Friendly Rate Control. In *Proc. IFIP/IEEE MMNS2001*, pages 29–42, 2001.

[33] H. Wu, M. Claypool, and R. Kinicki. A Model for MPEG with Forward Error Correction and TCP-Friendly Bandwidth. In *Proc. NOSSDAV2003*, June 2003.

[34] V. Paxson. "End-to-End Internet Packet Dynamics," *IEEE/ACM Trransactions on Networking*, Fall 1999.

[35] S. Futemma, K. Yamane, E. Itakura, “TFRC-based Rate Control Scheme for Real-time JPEG 2000 Video Transmission,” *IEEE Consumer Communications and Networking Conference*, pages 539-543, January 2005.

[36] H. Song, J. Kim, and C.-C. Jay Kuo, “Improved H.263+ rate control via variable frame rate adjustment and hybrid I-frame coding,” *Proc. IEEE ICIP'98*, vol. 2, pp. 375-378, Oct. 1998.

[37] H. Song and C.-C Jay Kuo, “Rate control for Low-Bit-Rate Video via Variable-Encoding Frame Rates”, *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 4, pp. 512–521, April. 2001.

[38] T. Research, *TMN(H.263) encoder/decoder, version 2.0, tmn (h.263) codec*, June 1996.

[39] “H.263+ Encoder/Decoder,” Image Processing Lab, Univ. British Columbia, Canada, Feb. 1998. TMN(H.263) codec.

[40] *Video Coding for Low Bitrate Communication*, ITU-T Recommendation H.263 Version 2, Jan.1998.

[41] C. W. Wong, O. C. Au, R. C.-W. Wong, and H.-K. Lam, “Real-time rate control via variable frame rate and quantization parameters,” *Proc. PCM 2004*, LNCS 3333, pp. 314-322, Oct. 2004.

[42] V. Baroncini, R. Felice, and G. Iacovoni, “Variable frame rate control jerkiness-driven,” *J. Real-Time Image Proc.*, vol. 4, no. 2, pp. 167–179, Jun. 2009.

[43] D. E. Comer, *Internetworking with TCP/IP*, vol. 1, 5th edition, Prentice Hall, 2005.

[44] J. Lee and B. W. Dickinson, “Temporally adaptive motion interpolation exploiting temporal masking in visual perception,” *IEEE Trans. Image Processing*, vol. 3, pp. 513–526, Sept. 1994.

[45] H. Song, J. Kim, and C. C. J. Kuo, “Real-time encoding frame rate control for H.263+ video over the Internet,” *Signal Processing: Image Commun.*, vol. 15, no. 1–2, pp. 127–148, 1999.

[46] L. Merritt and R. Vanam, “Improved rate control and motion estimation for H.264 encoder,” *Proc. IEEE ICIP'07*, vol V, pp. 309-312, 2007.

[47] Max/MSP/Jitter third party external objects for broadcasting: http://www.nullmedium.de/

[48] Computer version for Jitter external objects (cv.jit library): http://www.iamas.ac.jp/~jovan02/cv/objects.html

[49] M. Antonini, M. Barlaud. P. Mathieu, and I.Daubechies, “Image coding using wavelet transform,” *IEEE Trans, Image Processing,* vol.1, pp.205-220, April 1992.

[50] Y. He, X. Zhao, S. Yang, Y. Zhong, “Variable frame-rate video coding Based on global motion analysis”, H.-Y. Shum, M. Liao, and S.-F. Chang (Eds.): *Proc. PCM* 2001, LNCS 2195, pp. 426–433, 2001.

[51] C. Guaragnella, E. D. Sciascio, “Variable frame rate for very low bit-rate video coding”, 10th Mediterranean Electrotechnical Conference, MeleCon, Vol.2, pp. 503-506, 2000.

[52] J. Kim, Y.Kim, H. Song, T. Kuo, Y. Chung, C.-C. J. Kuo, “TCP-Friendly Internet video streaming employing variable frame-rate encoding and interpolation”, *IEEE Trans. on Circuits and System for Video Technology*, Vol. 10, No.7, Octobler 2000, pp1164-1174.

[53] R. Qiao and M. H. Lee, “Adaptive rate control of Motion-JPEG2000 video over IP networks”, ATNAC 2007, pp 328-331, 2007.

[54] J. R. Cooperstock, “Interacting in Shared Reality,” *11th Intl. Conf. on Human- Computer Interaction*, Las Vegas, NV, Jul. 2005. www.cim.mcgill.ca/sre/publications/

[55] McGill Ultra-Videoconferencing System website: ultravideo.mcgill.edu

[56] Converts RGB to monochrome (luminance), *jit.rgb2luma* object reference online at: http://www.cycling74.com/docs/max5/refpages/jit-ref/jit.rgb2luma.html

[57] Jitter Matrix Operators (MOP) reference page available online at: http://www.cycling74.com/docs/max5/refpages/jit-ref/jit.group-mop.html

[58] Common Box Attributes: a list of attributes shared by all max/msp/jitter objects, online at: http://www.cycling74.com/docs/max5/refpages/max-ref/jbox.html

[59] J. T. McClave and T. Sincich, *Statistics*, 9th edition, Prentice Hall, 2003.

[60] V. Baroncini, R. Felice, and G. Iacovoni, “Variable frame rate control jerkiness-driven,” *J. Real-Time Image Proc.*, vol. 4, no. 2, pp. 167–179, Jun. 2009.

[61] M. M. Taylor and C. D. Creelman, “PEST: Efficient estimates on probability functions,” *J. Acoust. Society America*, vol. 41, pp. 782–787, 1967.

[62] *Methodology for the subjective assessment of the quality of television pictures*, Rec. ITU-R BT.500-11, 2002.

[63] http://www.videolan.org/developers/x264.html

[64] International Telegraph and Telephone Consultative Committee (CCITT), *Progressive Bi-level Image Compression*, Recommendation T.82. February 1992

[65] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard", *IEEE Transactions. Circuits and Systems for Video Technology*, col.13. no.7, pp.560-576, July 2003

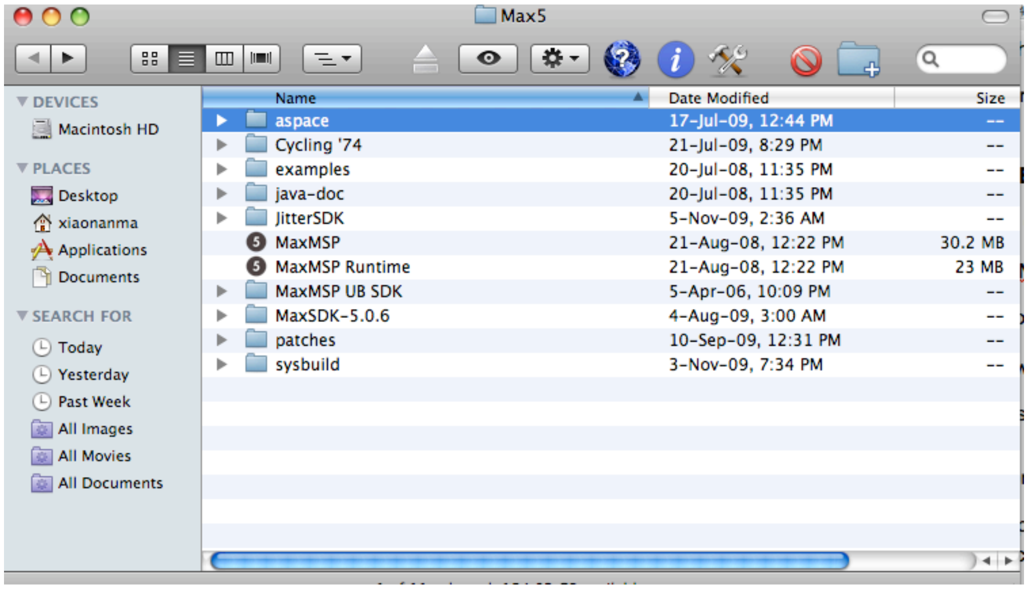
[66] O.Matthes, shoucast~ [Online] Available at www.maxobjects.com

[67] Ogg/Vorbis streaming externals for Max/MSP, [Online] Available at http://www.nullmedium.de/dev/oggpro/

[68] netsend~ for Max/MSP and Pure Data, [Online] Available at http://www.nullmedium.de/dev/netsend~/

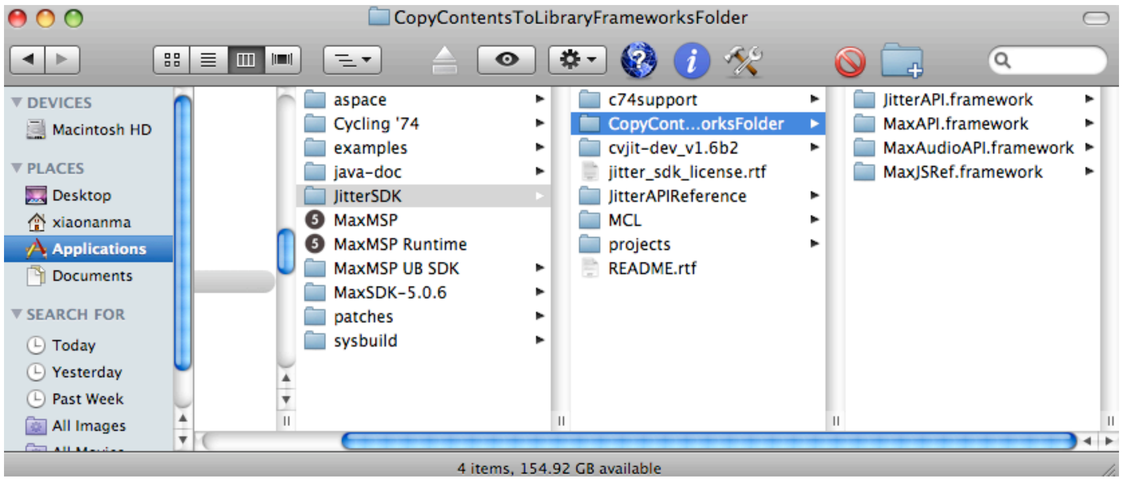
Appendix 1: Compiling Jitter Externals Under Mac OS X

To get started with Jitter object projects, one needs to download the Jitter SDK from the Max/MSP/Jitter website [12], and move the JitterSDK file into the Max folder, as shown below.



Name	Date Modified	Size
aspace	17-Jul-09, 12:44 PM	--
Cycling '74	21-Jul-09, 8:29 PM	--
examples	20-Jul-08, 11:35 PM	--
java-doc	20-Jul-08, 11:35 PM	--
JitterSDK	5-Nov-09, 2:36 AM	--
MaxMSP	21-Aug-08, 12:22 PM	30.2 MB
MaxMSP Runtime	21-Aug-08, 12:22 PM	23 MB
MaxMSP UB SDK	5-Apr-06, 10:09 PM	--
MaxSDK-5.0.6	4-Aug-09, 3:00 AM	--
patches	10-Sep-09, 12:31 PM	--
sysbuild	3-Nov-09, 7:34 PM	--

In the JitterSDK folder, there is a "Copy Contents To Library Frameworks Folder," the contents of which should be dragged to the Library/Frameworks directory.



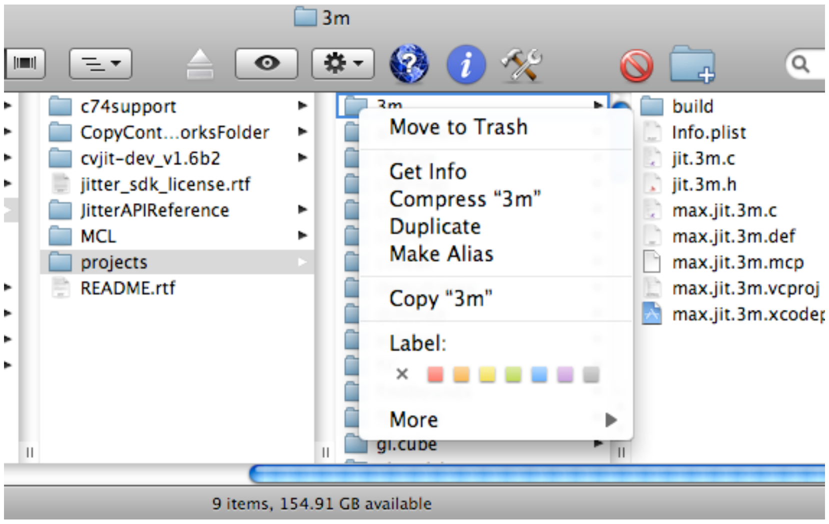
aspace
Cycling '74
examples
java-doc
JitterSDK
MaxMSP
MaxMSP Runtime
MaxMSP UB SDK
MaxSDK-5.0.6
patches
sysbuild

c74support
CopyCont...orksFolder
cvjit-dev_v1.6b2
jitter_sdk_license.rtf
JitterAPIReference
MCL
projects
README.rtf

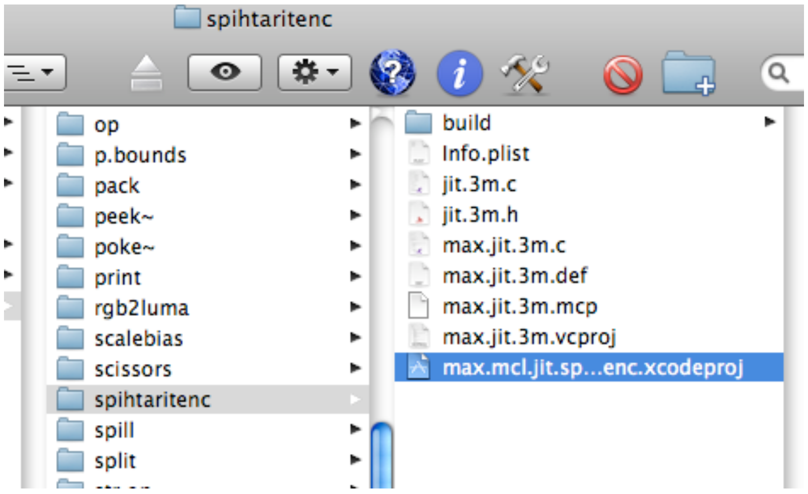
JitterAPI.framework
MaxAPI.framework
MaxAudioAPI.framework
MaxJSRef.framework

4 items, 154.92 GB available

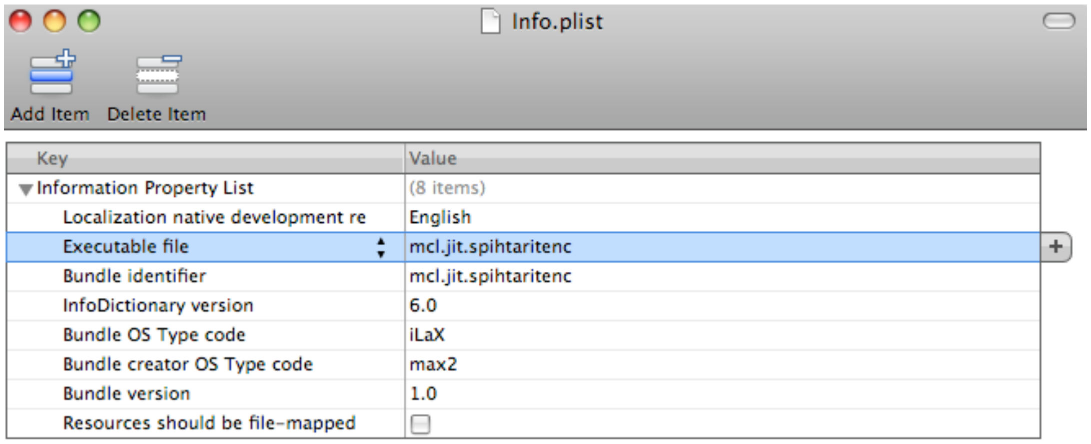
Now the Jitter SDK has been installed and ready to use. To create one's own project, one could simply copy an existing project into the JitterSDK folder and rename the folder to the desired object's name. In this example, we will compile an external object called **mcl.jit.spihtaritenc.**



The easiest way to start is to copy any existing project (here we copy the project corresponding to the standard object "3m") and rename the folder to the desired name, in our case "spihtaritenc." Change the project name from "max.jit.3m.xcodeproj" to "max.mcl.jit.spihtaritenc.xcodeproj."

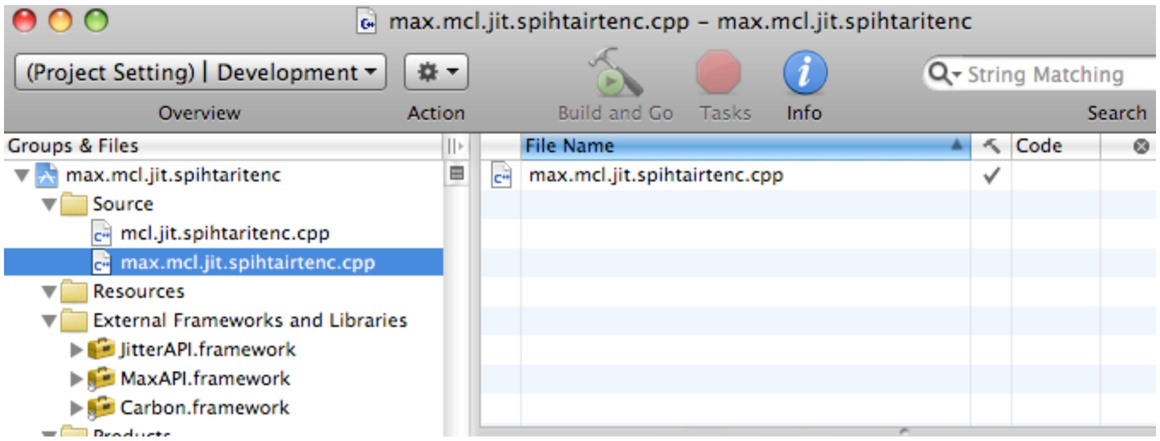


Next, open the “info.plist” file, change both executable file and bundle identifier to “mcl.jit.spihtaritenc”.

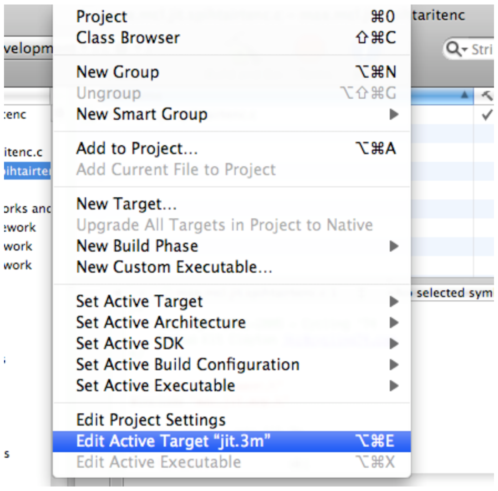


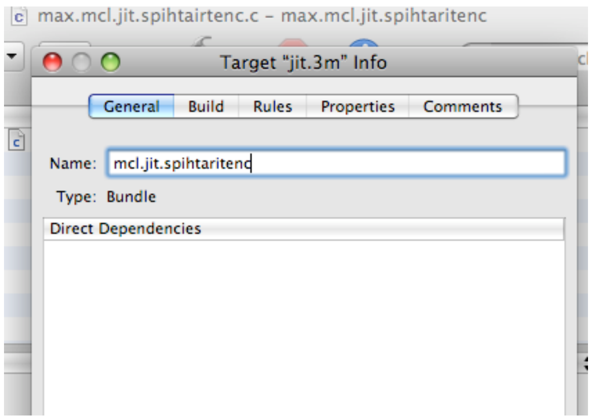
Key	Value
Information Property List	(8 items)
Localization native development re	English
Executable file	mcl.jit.spihtaritenc
Bundle identifier	mcl.jit.spihtaritenc
InfoDictionary version	6.0
Bundle OS Type code	iLaX
Bundle creator OS Type code	max2
Bundle version	1.0
Resources should be file-mapped	☐

Now double click on the xcodeproj file to open the whole project, and rename two source files as shown in the figure below.



Click on the Project → Edit Active Target “jit.3m”, change its name to “mcl.jit.spihtaritenc” under “General” tab.




max.mcl.jit.spihtairtenc.c – max.mcl.jit.spihtaritenc

Target "jit.3m" Info

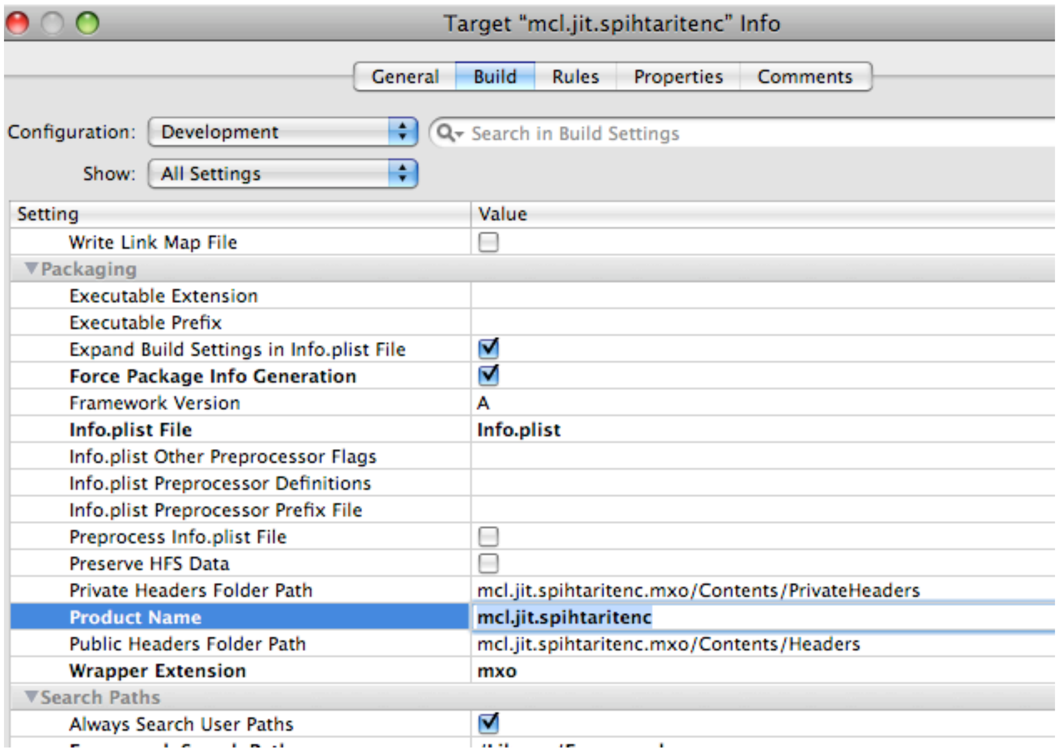
General | Build | Rules | Properties | Comments

Name: mcl.jit.spihtaritenc

Type: Bundle

Direct Dependencies

Go to "Build" list, change "Product Name" to mcl.jit.spihtaritenc for both development and deployment configuration.


Target "mcl.jit.spihtaritenc" Info

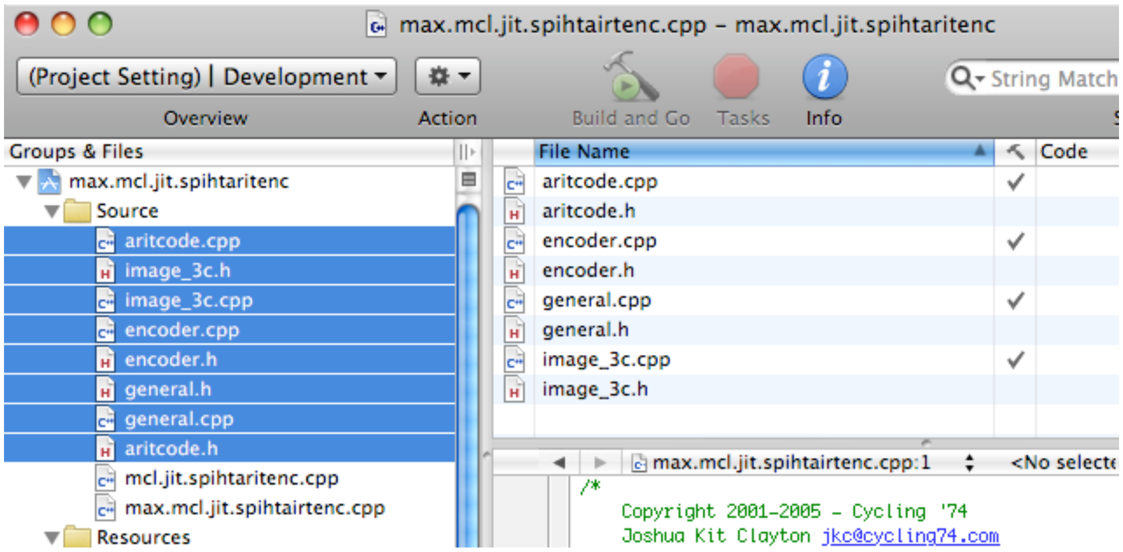
General | Build | Rules | Properties | Comments

Configuration: Development

Show: All Settings

Setting	Value
Write Link Map File	☐
▼Packaging	
Executable Extension	
Executable Prefix	
Expand Build Settings in Info.plist File	✓
Force Package Info Generation	✓
Framework Version	A
Info.plist File	**Info.plist**
Info.plist Other Preprocessor Flags	
Info.plist Preprocessor Definitions	
Info.plist Preprocessor Prefix File	
Preprocess Info.plist File	☐
Preserve HFS Data	☐
Private Headers Folder Path	mcl.jit.spihtaritenc.mxo/Contents/PrivateHeaders
Product Name	**mcl.jit.spihtaritenc**
Public Headers Folder Path	mcl.jit.spihtaritenc.mxo/Contents/Headers
Wrapper Extension	**mxo**
▼Search Paths	
Always Search User Paths	✓

The next step is to replace the old C/C++ source files by the new source files, in our case the files that implement SPIHT encoder. As can be seen in any example project, there is always a "max.jit." file and a "jit." file. The "max.mcl.jit.spihtarit.cpp" is a Jitter wrapper file to import the code in "mcl.jit.spihtarit.cpp" and compile it as a Max external object. One needs to open both source files and replace every word containing “jit_3m” by “mcl_jit_spihtaritenc.” Now the executable object "mcl.jit.spihtaritenc" is ready to be built. One could simply drag all our SPIHT C++ codes to the source folder as shown in the figure below, and modify "mcl.jit.spihtarit" to call the main function of the original C++ source files.



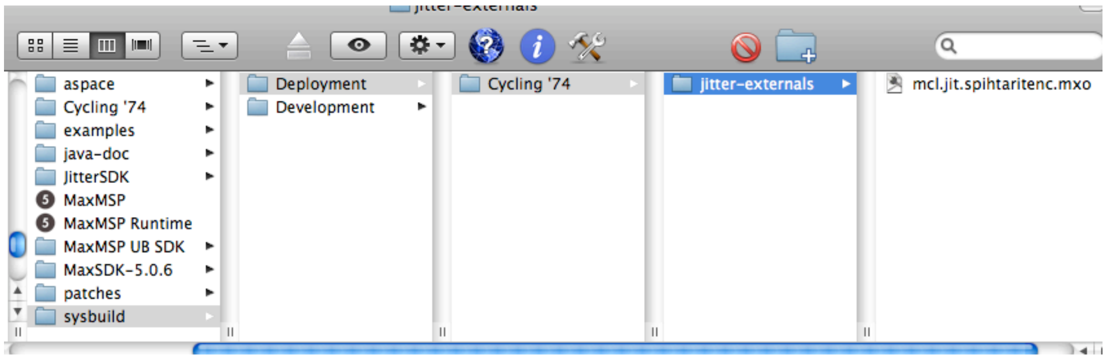
max.mcl.jit.spihtaritenc
Source
aritcode.cpp
image_3c.h
image_3c.cpp
encoder.cpp
encoder.h
general.h
general.cpp
aritcode.h
mcl.jit.spihtaritenc.cpp
max.mcl.jit.spihtairtenc.cpp
Resources

File Name	Code
aritcode.cpp	✓
aritcode.h	
encoder.cpp	✓
encoder.h	
general.cpp	✓
general.h	
image_3c.cpp	✓
image_3c.h	

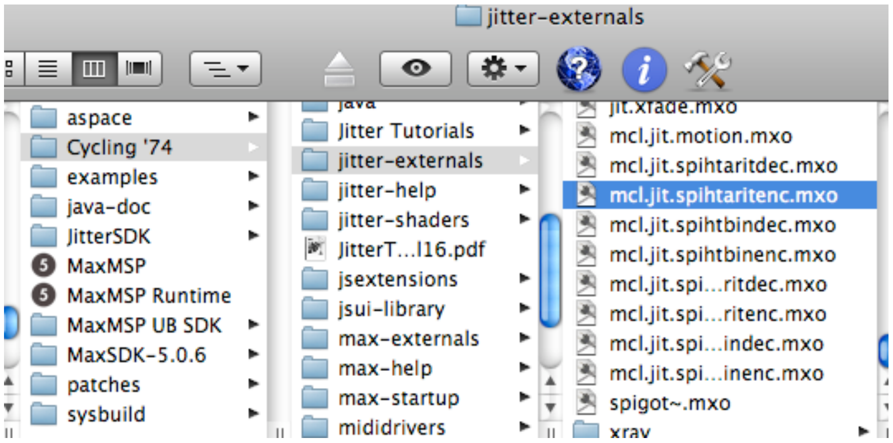
max.mcl.jit.spihtairtenc.cpp:1

```
/*
  Copyright 2001-2005 - Cycling '74
  Joshua Kit Clayton jkc@cycling74.com
```

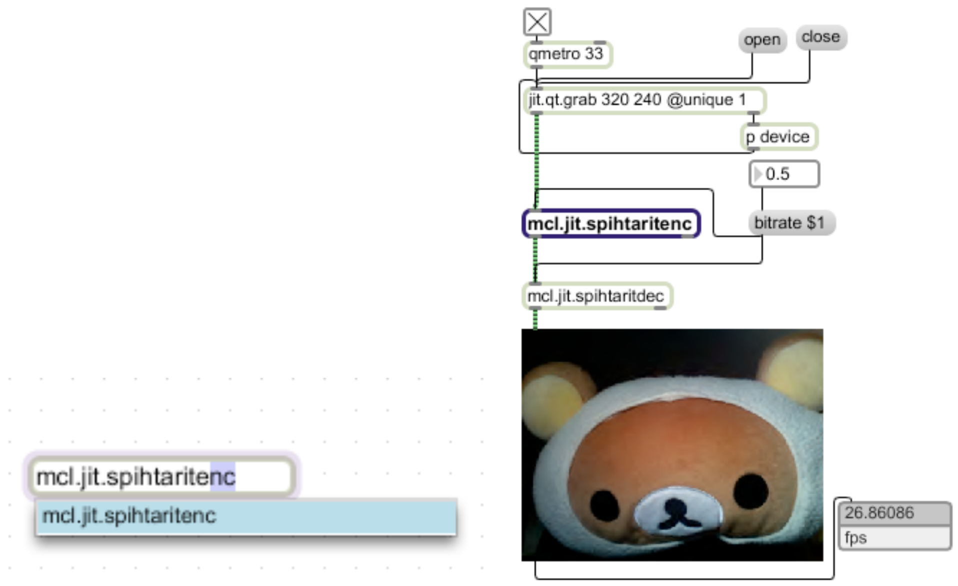
Normally, while developing the code, the building of the executable object is done in the “Development” configuration to reduce compiling time. However, when the code is ready, polished, and debugged, one should build the project in the “Deployment” configuration. This way, the resulting external object will run much faster.



The next step is to move the generated external object into the Cycling‘74/jitter-externals/ folder, where it can be seen by Max/MSP/Jitter, and make sure that there are no other objects bearing the same name on the computer's hard drive.



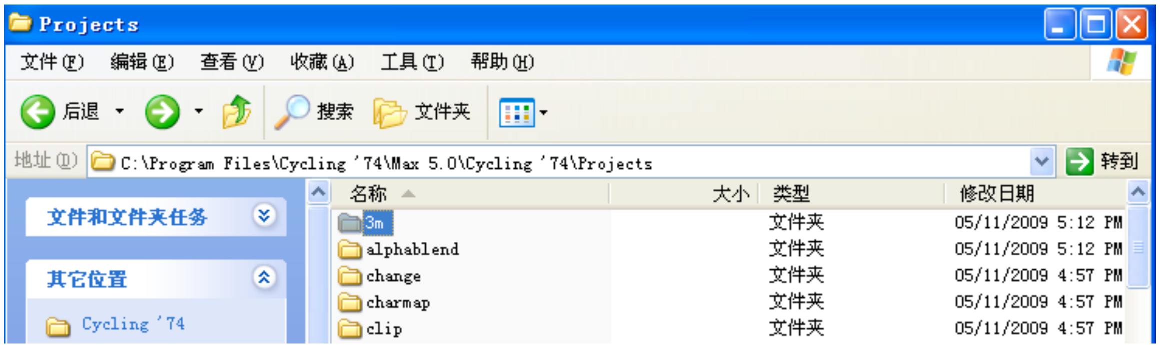
To test the external object, simply create a new patch in Max/MSP/Jitter and add the new object. Connect the object to the appropriate input(s) and output(s). In our example, we connect the input to **jit.qt.grab** for live video frame grabbing, and the output to a similarly generated SPIHT decoder object (**mcl.jit.spihtaritdec**), so that frames can be decoded and displayed, as shown in the figure below.



The compressed bit stream is exported from the encoder as a Jitter matrix (note the green chord connecting the encoder to the decoder), which enables it to be handled by other Jitter objects, for example **jit.net.send** and **jit.net.recv**.

Appendix 2: Compiling Jitter Externals Under Windows

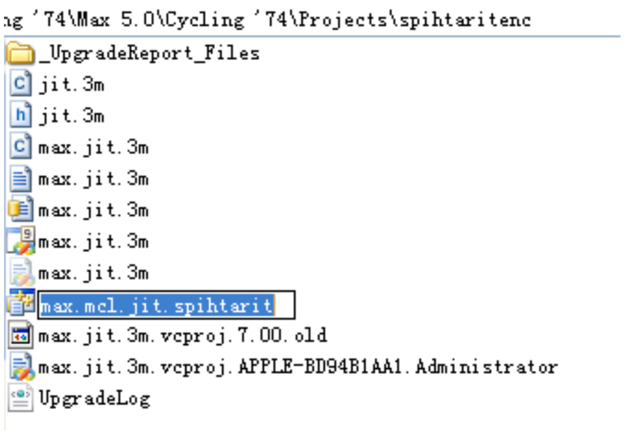
The Jitter SDK for Windows is different from the Mac version, and it is also available for download from [12]. The SDK package needs to be uncompressed and the projects content folders need to be put under the Max/MSP/Jitter directory, usually "C:\Program Files\Cycling ’74 \Max 5.0\Cycling ’74." The Frameworks folder needs to be copied as instructed in the previous section for Mac OS X. Then, one needs to open the Projects folder, choose an existing project, and rename it to the desired external object’s name. In the figure below, for example, we copy the “3m” folder and rename it to “spihtaritenc”.



C:\Program Files\Cycling ’74\Max 5.0\Cycling ’74\Projects

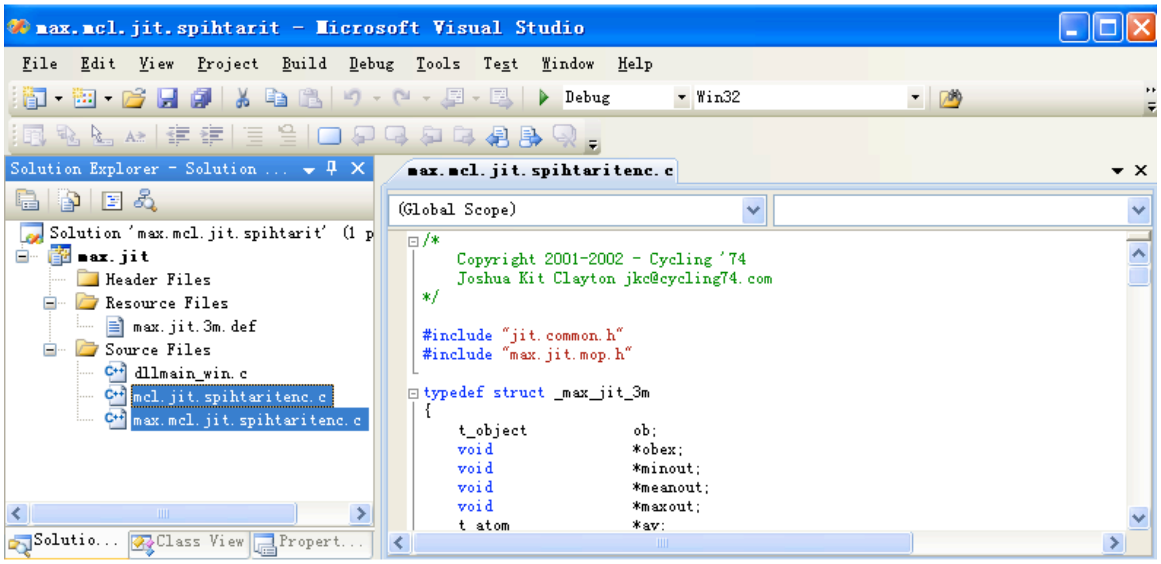
名称	大小	类型	修改日期
3m		文件夹	05/11/2009 5:12 PM
alphablend		文件夹	05/11/2009 5:12 PM
change		文件夹	05/11/2009 4:57 PM
charmap		文件夹	05/11/2009 4:57 PM
clip		文件夹	05/11/2009 4:57 PM

Open the newly renamed folder (in our case, "spihtaritenc"), and rename the main project file ("max.jit.3m") to the desired name (in our case "max.mcl.jit.spihtaritenc").



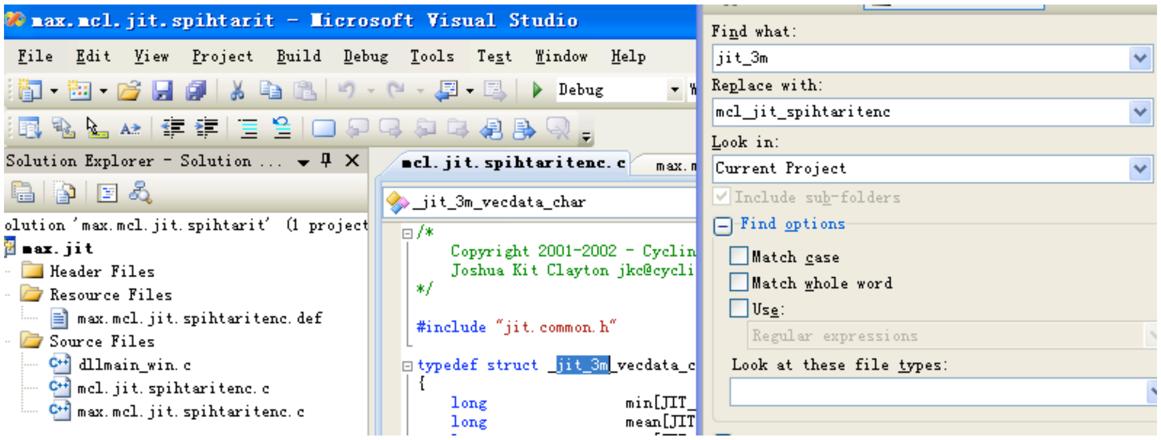
```
ng ’74\Max 5.0\Cycling ’74\Projects\spihtaritenc
_UpgradeReport_Files
jit.3m
jit.3m
max.jit.3m
max.jit.3m
max.jit.3m
max.jit.3m
max.jit.3m
max.mcl.jit.spihtarit
max.jit.3m.vcproj.7.00.old
max.jit.3m.vcproj.APPLE-BD94B1AA1.Administrator
UpgradeLog
```

Open the newly renamed project and rename the C/C++ files "max.jit.3m.c" and "jit.3m.c" to the desired names, in our case "max.mcl.jit.spihtaritenc.c" and "mcl.jit.spihtaritenc.c."


max.mcl.jit.spihtarit - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Help
Debug
Win32
Solution Explorer - Solution ...
Solution 'max.mcl.jit.spihtarit' (1 p
max.jit
Header Files
Resource Files
max.jit.3m.def
Source Files
dllmain_win.c
mcl.jit.spihtaritenc.c
max.mcl.jit.spihtaritenc.c
Solutio...
Class View
Propert...
max.mcl.jit.spihtaritenc.c
(Global Scope)

```
/*
    Copyright 2001-2002 - Cycling '74
    Joshua Kit Clayton jkc@cycling74.com
*/

#include "jit.common.h"
#include "max.jit.mop.h"

typedef struct _max_jit_3m
{
    t_object        ob;
    void            *obex;
    void            *minout;
    void            *meanout;
    void            *maxout;
    t_atom          *av;
```

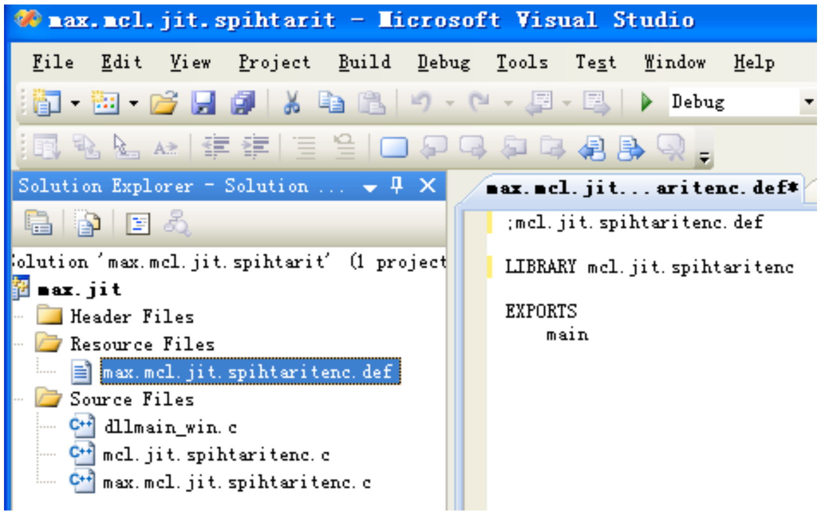
Also replace all occurrences of "jit.3m" by the desired name ("mcl.jit.spihtaritenc") in those two files.


max.mcl.jit.spihtarit - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Help
Debug
Solution Explorer - Solution ...
olution 'max.mcl.jit.spihtarit' (1 project
max.jit
Header Files
Resource Files
max.mcl.jit.spihtaritenc.def
Source Files
dllmain_win.c
mcl.jit.spihtaritenc.c
max.mcl.jit.spihtaritenc.c
mcl.jit.spihtaritenc.c
max.m
_jit_3m_vecdata_char
Find options
Regular expressions

Find what: jit_3m
Replace with: mcl_jit_spihtaritenc
Look in: Current Project
[✓ Include sub-folders]
Match case
Match whole word
Use:
Look at these file types:

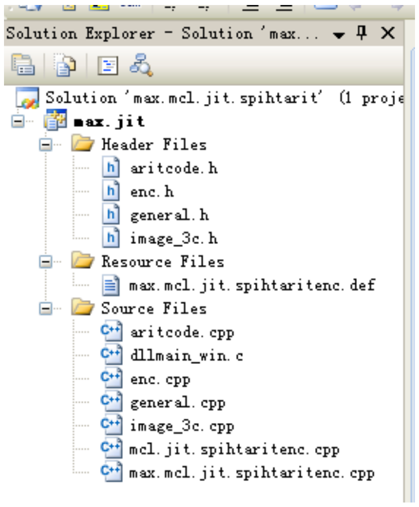
```
/*
    Copyright 2001-2002 - Cyclin
    Joshua Kit Clayton jkc@cycli
*/

#include "jit.common.h"

typedef struct _jit_3m_vecdata_c
{
    long            min[JIT_
    long            mean[JIT
```

In all Windows SDK projects, there is a definition (".def") file among the resource files (see figure below). One can simply use the existing file (after renaming it appropriately) and replace all occurrences of "jit.3m" by the desired project name (in our case, "mcl.jit.spihtaritenc") in the file content.

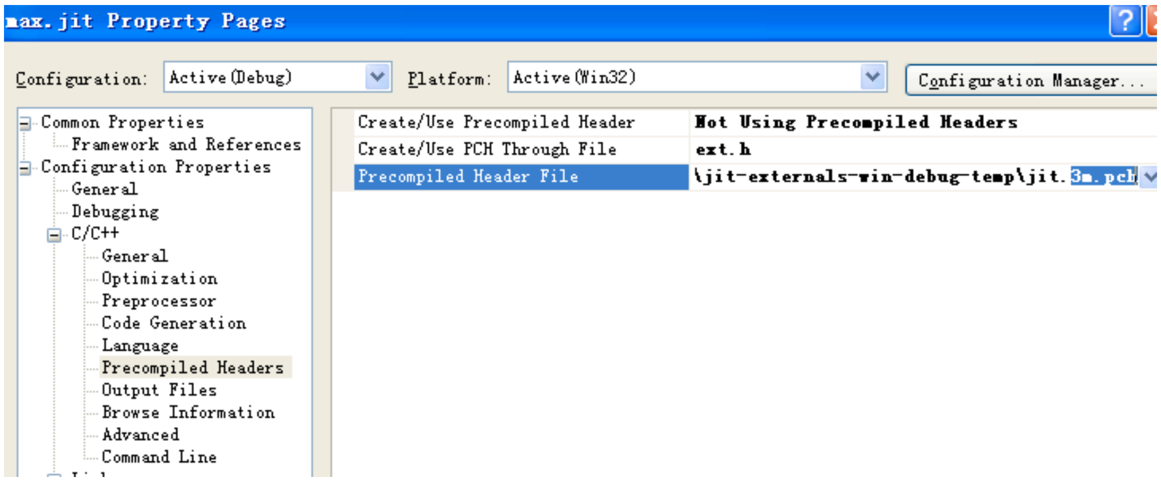

max.mcl.jit.spihtarit - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Help
Debug
Solution Explorer - Solution ...
max.mcl.jit...aritenc.def*
Solution 'max.mcl.jit.spihtarit' (1 project
max.jit
Header Files
Resource Files
max.mcl.jit.spihtaritenc.def
Source Files
dllmain_win.c
mcl.jit.spihtaritenc.c
max.mcl.jit.spihtaritenc.c

```
;mcl.jit.spihtaritenc.def

LIBRARY mcl.jit.spihtaritenc

EXPORTS
    main
```

Next, the C/C++ files with the desired code should be included into the project. In our case, we include SPIHT encoder’s C++ files, as shown below.



Open Project → max.jit Properties and click on Configuration Properties List. Find Precompiled Headers under C/C++ list, and change "jit.3m.pch" to the appropriate name ("mcl.jit.spihtaritenc.pch").

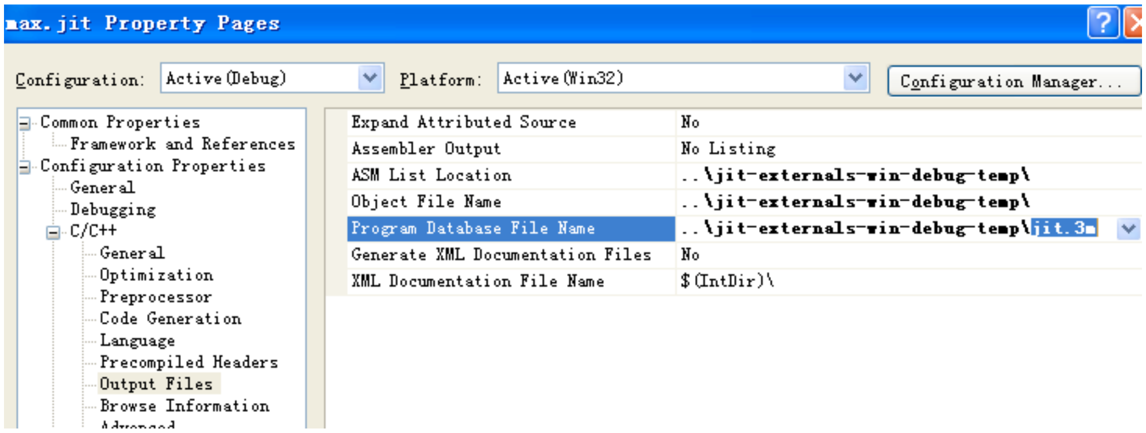

max.jit Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

- Common Properties
  - Framework and References
- Configuration Properties
  - General
  - Debugging
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - Command Line

Create/Use Precompiled Header	Not Using Precompiled Headers
Create/Use PCH Through File	ext.h
Precompiled Header File	\jit-externals-win-debug-temp\jit.3m.pch

Next, change object name for Program Database File Name under C/C++ → Output Files.

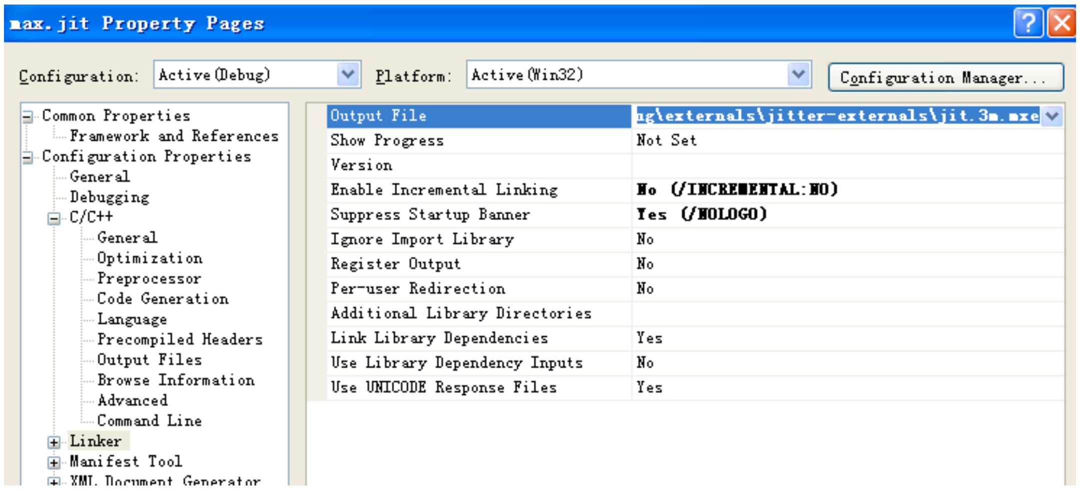

max.jit Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

- Common Properties
  - Framework and References
- Configuration Properties
  - General
  - Debugging
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information

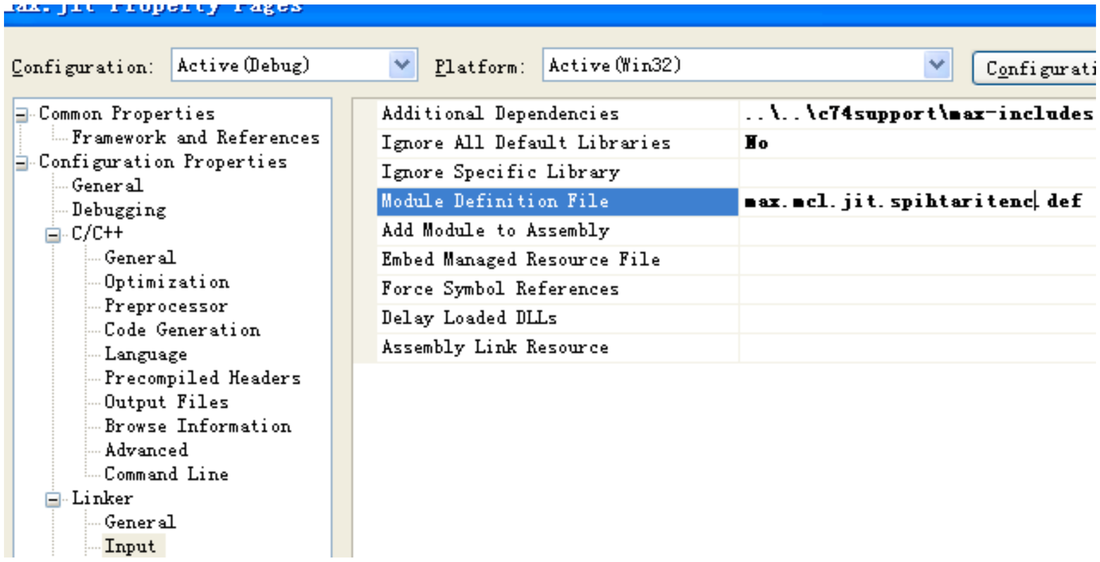
Expand Attributed Source	No
Assembler Output	No Listing
ASM List Location	..\jit-externals-win-debug-temp\
Object File Name	..\jit-externals-win-debug-temp\
Program Database File Name	..\jit-externals-win-debug-temp\jit.3m
Generate XML Documentation Files	No
XML Documentation File Name	$(IntDir)\

Also, "jit.3m" needs to be replaced by the appropriate name (in our case, "mcl.jit.spihtaritenc") for Linker and Input in the list, as shown in figures below.



Configuration: Active (Debug) Platform: Active (Win32)

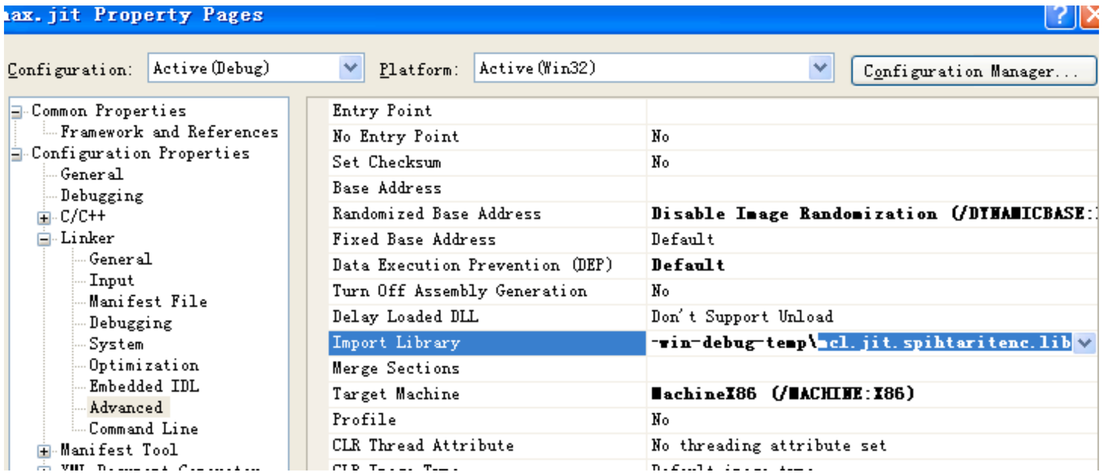
Output File	ng\externals\jitter-externals\jit.3m.mxe
Show Progress	Not Set
Version	
Enable Incremental Linking	No (/INCREMENTAL:NO)
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes



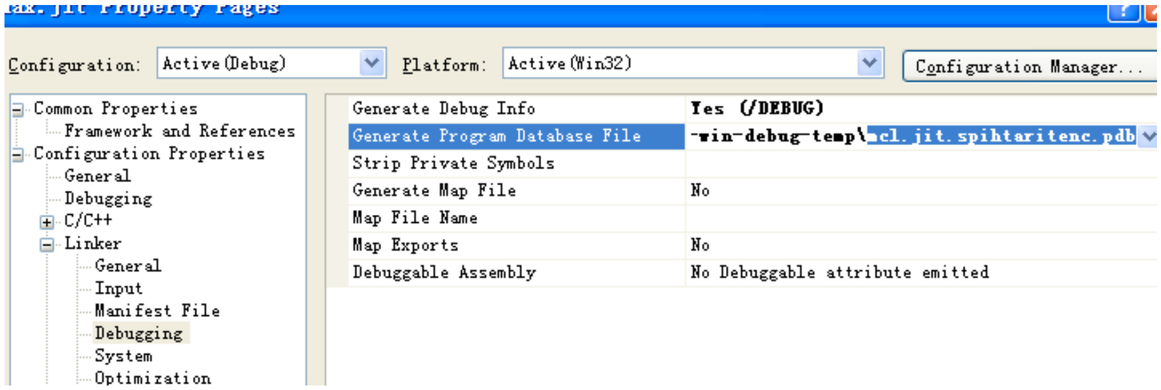
Configuration: Active (Debug) Platform: Active (Win32)

Additional Dependencies	..\..\c74support\max-includes
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	max.mcl.jit.spihtaritenc.def
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

The same replacements need to be made for Import Library under Advanced and “Generate Program Database File” under “Debugging.”

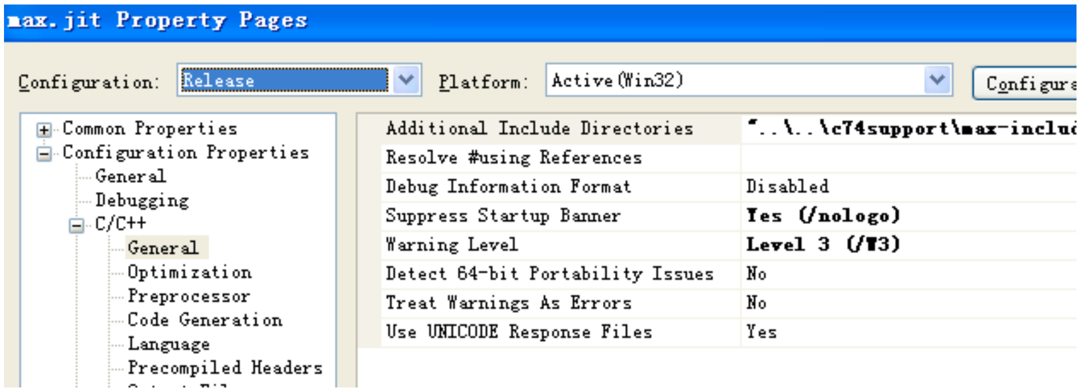


Entry Point	
No Entry Point	No
Set Checksum	No
Base Address	
Randomized Base Address	**Disable Image Randomization (/DYNAMICBASE:**
Fixed Base Address	Default
Data Execution Prevention (DEP)	**Default**
Turn Off Assembly Generation	No
Delay Loaded DLL	Don't Support Unload
Import Library	**-win-debug-temp\cl.jit.spihtaritenc.lib**
Merge Sections	
Target Machine	**MachineX86 (/MACHINE:X86)**
Profile	No
CLR Thread Attribute	No threading attribute set



Generate Debug Info	**Yes (/DEBUG)**
Generate Program Database File	**-win-debug-temp\cl.jit.spihtaritenc.pdb**
Strip Private Symbols	
Generate Map File	No
Map File Name	
Map Exports	No
Debuggable Assembly	No Debuggable attribute emitted

Exactly the same changes need to be made to the project properties for “Release” compiling configuration, which plays the role of "Deployment" configuration in Xcode on Mac OS.

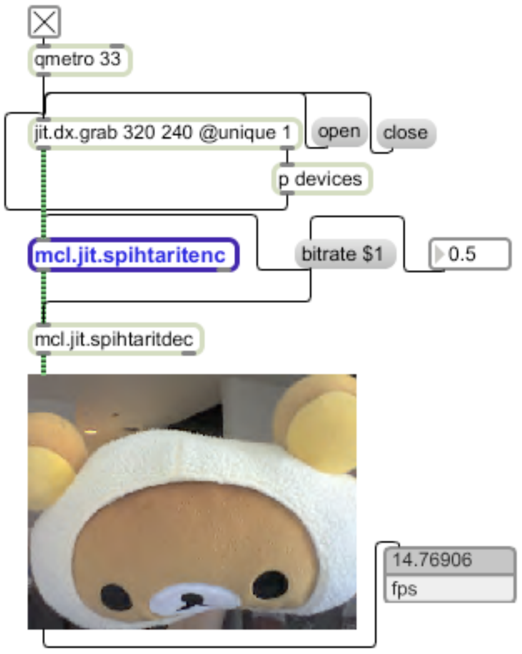

max.jit Property Pages

Configuration: Release Platform: Active(Win32) Configure

Additional Include Directories	"..\..\c74support\max-includ
Resolve #using References	
Debug Information Format	Disabled
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level 3 (/W3)
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Now the project is ready to compile. The compiled external objects will be located in C:\Program Files\Cycling '74\Max 5.0\Cycling '74\sysbuild\ sin_realease\externals\jitter-externals. One should then cut and paste them into C:\Program Files\Cycling '74\ Max 5.0\Cycling '74\jitter-externals, and make sure this is the only copy of the external object on the hard drive.

To test the compiled external object, create a new patch in Jitter, include the newly created external object (in our case, **mcl.jit.spihtaritenc**) and connect it to the appropriate input(s) and output(s). In our example, we connect the input to **jit.dx.grab** for live frame grabbing, and the output to the SPIHT decoder for decoding and display, as shown below. The **jit.dx.grab** object grabs frames from an external source using DirectX, and is for use on Windows machines only. It is equivalent to **jit.qt.grab** on Mac OS.


qmetro 33
jit.dx.grab 320 240 @unique 1
open
close
p devices
mcl.jit.spihtaritenc
bitrate $1
0.5
mcl.jit.spihtaritdec
14.76906
fps

Appendix 3: Interfacing One's Code with Jitter Environment

By looking at the source code of objects supplied with the SDK, one can find that interfacing of different objects with the Jitter environment is very similar. Usually, similar modifications to one's own code will enable it to interface with the Jitter environment. The steps for making these changes are the same for Mac and Windows systems, and will be described in this section.

Max Wrappers

A Max wrapper class is defined to expose a Jitter object to the Max patch. A basic Max wrapper will require the following:

1) `typedef struct _max_mcl_jit_spihtaritenc`
2) `int main(void)`
3) `static void *max_mcl_jit_spihtaritenc_new`
4) `void max_mcl_jit_spihtaritenc_free`

If the existing project is written in C, one needs to add the following before the main function.

```
#ifdef __cplusplus
         extern "C"
 #endif
```

The first step is to define the object's class structure. In our example of the SPIHT encoder, we used the following definition.

```
    typedef struct _max_mcl_jit_spihtaritenc
    {
     t_object          ob;
     void              *obex;
    } t_max_mcl_jit_spihtaritenc;
```

The `t_object` type is essential for all Max objects as the first entry in the object's structure. The `obex` is a pointer to external information that will be processed by the Jitter object. The attribute information and resources for inlets (inputs) and outlets (outputs) of the object are all stored in the “obex” data.

The definition of our Max class is in the main function, as shown below.

```
int main(void)
{
      void *p,*q;
      mcl_jit_spihtaritenc_init();   // initialize the Jitter class

    // create the Max class as documented in Writing Max Externals
    setup((t_messlist **)&max_mcl_jit_spihtaritenc_class,
(method)max_mcl_jit_spihtaritenc_new,
(method)max_mcl_jit_spihtaritenc_free,
(short)sizeof(t_max_mcl_jit_spihtaritenc), 0L, A_GIMME, 0);

    // specify a byte offset to keep additional information
      p =
max_jit_classex_setup(calcoffset(t_max_mcl_jit_spihtaritenc,obex));
    // look up the Jitter class in the class registry
```

```
      q = jit_class_findbyname(gensym("mcl_jit_spihtaritenc"));

     max_jit_classex_mop_wrap(p,q,0);
    // wrap the Jitter class with the standard methods for Jitter
objects
     max_jit_classex_standard_wrap(p,q,0);

    // add an inlet/outlet assistance method
    addmess((method)max_jit_mop_assist, "assist", A_CANT,0);

      return 0;
    }
```

Now that the object's class is wrapped, one can start adding additional methods such as inlet and outlet assistance functions, as needed.

The third essential function in Max wrapper class is the constructor `*max_mcl_jit_spihtaritenc_new` function, as shown below:

```

static void *max_mcl_jit_spihtaritenc_new(t_symbol *s, long argc, t_atom
*argv)
{
      t_max_mcl_jit_spihtaritenc *x ;
      void *o,*m;//,*mop;
      t_jit_matrix_info info;

// create the wrapper object instance based on the max wrapper class,
and the jitter class
      if (x=(t_max_mcl_jit_spihtaritenc *)
max_jit_obex_new(max_mcl_jit_spihtaritenc_class,gensym
("mcl_jit_spihtaritenc"))) {
```

```
            // instantiate Jitter object
            if (o=jit_object_new(gensym("mcl_jit_spihtaritenc"))) {

           // Define output matrix specially for SPIHT encoder
            max_jit_mop_setup_simple(x,o,argc,argv);

            m = max_jit_mop_getoutput(x,1);

            jit_object_method(m,_jit_sym_getinfo,&info);
                  info.type         = _jit_sym_char;
                  info.planecount = 1;
                  info.dimcount     = 2;
                  jit_object_method(m,_jit_sym_setinfo,&info);

            // process attribute argument
                  max_jit_attr_args(x,argc,argv);
            } else {
                // couldn't instantiate, clean up and report an error
                  error("jit.spihtaritenc: could not allocate object");
                  freeobject((t_object*)x);
            }
      }
      return (x);
          }
```

As most Jitter objects output four-plane ARGB matrices, we need to define the format for SPIHT encoder separately. For a matrix operator object, we call `max_jit_mop_setup_simple` to define the properties of matrices. The data type is defined as `char` and the object output matrix is defined as single plane with 2 dimensions. This means that our SPIHT encoder will output one 2-D matrix of elements of type `char`, and the bits of these elements will be filled with the compressed bit stream.

After this, we can proceed to the last essential function acting as a destructor: `max_mcl_jit_spihtaritenc_free`.

```
void max_mcl_jit_spihtaritenc_free(t_max_mcl_jit_spihtaritenc *x)
{
      max_jit_mop_free(x);
      jit_object_free(max_jit_obex_jitob_get(x));
      max_jit_obex_free(x);
          }
```

The matrix operators require to call the `max_jit_mop_free(x)` to free the resources allocated for matrix inputs and outputs. The `jit_object_free` function will look up the internal Jitter object instance and free its resources. The `max_jit_obex_free(x)` function is to free resources related to the `obex` entry.

Defining a Jitter Class

The mcl.jit.spihtaritenc file will contain the definition of the object's Jitter class. A minimal Jitter class definition needs four basic elements:

1) `typedef struct _mcl_jit_spihtaritenc`

2) `t_jit_err mcl_jit_spihtaritenc_init(void)`

3) `t_mcl_jit_spihtaritenc *mcl_jit_spihtaritenc_new (void)`

4) `mcl_jit_spihtaritenc_free(t_mcl_jit_spihtaritenc *x)`

The first element `struct _mcl_jit_spihtaritenc` defines the class structure. We have to put `t_object` as the first field in the class structure definition. Since our SPIHT encoder will have one additional input (besides the input frame) that allows the user to specify the encoding bit rate, it can be added as an attribute to the Jitter object in mcl.jit.spihtaritenc file, as shown below:

```
typedef struct _mcl_jit_spihtaritenc
{
    t_object            ob;
    double              bitrate;

} t_mcl_jit_spihtaritenc;
```

The second element `mcl_jit_spihtaritenc_init` will initialize the object class. In the initialization function, a `jit_class_new` function, will call the class definition to start, followed by `jit_class_addmethod` and `jit_class_addattr` to register methods and their names to the object class.

```
t_jit_err mcl_jit_spihtaritenc_init(void)
{
      long attrflags=0;
      t_jit_object *attr, *mop,*o;

      _mcl_jit_spihtaritenc_class = jit_class_new
    ("mcl_jit_spihtaritenc",               //create a class with its
name
    (method)mcl_jit_spihtaritenc_new,     //constructor
    (method)mcl_jit_spihtaritenc_free,    //destructor
    sizeof(t_mcl_jit_spihtaritenc),
    0L);
```

The input bit rate was registered as an attribute to the object class as shown below.

```
attr =jit_object_new
      (_jit_sym_jit_attr_offset,     //instantiate an obejct
       "bitrate",                    //with name “bitrate”
       _jit_sym_float64,            //type float64
       attrflags,                   //default flags
       (method)0L,                  //default getter accessor
       (method)0L,                  //default setter accessor
calcoffset(t_mcl_jit_spihtaritenc,bitrate)); //byte offset to struct
member
```

This was followed with `jit_class_addattr` and `jit_class_register` functions to add and register the bit rate attribute to spihtaritenc class.

```
    jit_class_addattr(_mcl_jit_spihtaritenc_class,attr);

    jit_class_register(_mcl_jit_spihtaritenc_class);
```

In the `mcl_jit_spihtaritenc_init` function, we also need to create the matrix operator (mop) for the object class. Mop can process the matrix type and it is needed for all Jitter objects dealing with video frames. The mop can be defined as shown below:

```
    //add mop
    // create a new instance of jit_mop with 1 input, and 1 output
     mop = (t_object*)jit_object_new(_jit_sym_jit_mop,1,1);

     // enforce a single type for all inputs and outputs
     jit_mop_single_type(mop,_jit_sym_char);

     // add the jit_mop object as an adornment to the class

     jit_class_addadornment(_mcl_jit_spihtaritenc_class,mop);
```

The mop will be bound to the symbol `matrix_calc` in `jit_class_addmethod` function as a private, untyped method with `A_CANT` type signature. An example of `jit_class_addmethod` function in our mcl.jit.spihtaritenc file is shown below.

```
//add methods
jit_class_addmethod(
_mcl_jit_spihtaritenc_class,                   // class pointer
(method)mcl_jit_spihtaritenc_matrix_cal        // call function
"matrix_calc",                                //mehotd name
    A_CANT, 0L);                 //type signature for the method
```

The other two important methods that are required for all objects are the constructor and destructor functions: `*mcl_jit_spihtaritenc_new` and `mcl_jit_spihtaritenc_free`. In the constructor function, we need to allocate and initialize the object's structure as shown below:

```
t_mcl_jit_spihtaritenc *mcl_jit_spihtaritenc_new(void)
{
   t_mcl_jit_spihtaritenc *x;

  if
(x=(t_mcl_jit_spihtaritenc*)jit_object_alloc(_mcl_jit_spihtaritenc_class
))
   {
       x->bitrate = 0.5;
      } else {
           x = NULL;
      }
      return x;
    }
```

This means that if the allocation is successful, the `t_mcl_jit_spihtaritenc *x` will be initialized to a default bit rate value of 0.5 bits per pixel (bpp). With all four basic elements taken care of, the Jitter object can be compiled.

Now we can proceed to `mcl_jit_spihtaritenc_matrix_calc` function, where the matrix processing method is defined. In the `mcl_jit_spihtaritenc_matrix_calc` function, the object method will first read matrix information by the following code.

```
// get the zeroth index input and output from the corresponding input
and output lists
in_matrix = jit_object_method(inputs,_jit_sym_getindex,0);

out_matrix = jit_object_method(outputs,_jit_sym_getindex,0);
```

We need to lock access to input and output matrices and get matrix data pointers before the method can actually process the data. The matrix structures will be filled out for input and output after locking.

```
in_savelock = (long) jit_object_method(in_matrix,_jit_sym_lock,1);
out_savelock = (long) jit_object_method(out_matrix,_jit_sym_lock,1);

jit_object_method(in_matrix,_jit_sym_getinfo,&in_minfo);

jit_object_method(out_matrix,_jit_sym_getinfo,&out_minfo);
```

To define dimensions of SPIHT compressed data, we assigned `out_minfo.dim` as shown below:

```
   out_minfo.dim[1] = 1;

   out_minfo.dim[0] =
   bitrate*(in_minfo.dim[0]*in_minfo.dim[1])/8.0;
```

In this code, `in_minfo.dim` is the dimension of video inputs, and its value has been automatically read by `jit_object_method(m,_jit_sym_getinfo,&info)` function in the Max wrapper file max.mcl.jit.spihtaritenc.

In Jitter, matrices can have multiple planes. For example, a color video frame would typically have four planes: one for Alpha (transparency), and three for the color components (RGB). A pointer to the beginning of each row of the matrix is obtained by adding the corresponding byte stride, which is the number of bytes between the starting pixels of two consecutive rows. The byte strides are defined in the `t_jit_matrix_info` structure. An example from our code is shown below for a three-plane matrix (Alpha plane is not encoded by our SPIHT encoders).

```
    for (i=0;i<height;i++)
          {
          // increment our data pointers according to byte stride
                ip = in_bp + i*in_minfo->dimstride[1];
                      for (j=0;j<width;j++) {
                           *ip++;
                             img[k++] = *ip++;
                             img[k++] = *ip++;
                             img[k++] = *ip++;
                      }
          }
     encodermain(width, height, bitrate, &img[0], out_bp);
```

Structure `in_minfo` contains information about the input matrix of the Jitter object, while its start is pointed to by `in_bp`. After the loop above has executed, the pixels of the input matrix will be copied to a one-dimensional array `img[k]`, which is the way the SPIHT encoder is set up to accept input pixels. Subsequently, the main encoder function is called with the start of the `img[k]` array as one of its arguments.

The main SPIHT encoder function is located in the "encoder.cpp" file. Its last argument is the pointer (`*op`) to the generated output bit stream, which will be subsequently cast as a Jitter matrix. An excerpt from the "encoder.cpp" file is shown below. As shown, the first four bytes of the compressed bit stream (`cmp[k]`) store the width and height of the frame.

Appendix 4: Electronic files

Electronic data and files listed below and appended as supplemental files or as a CD or DVD, form part of this work under the copyright of this author.

Projects files inside “mcl.jit_MacOSX” folder were deployed using Xcode, and project files inside “mcl.jit_Windows” folder were deployed using Microsoft Visual C++.

Data File:

• mcl.jit_MacOSX 18.6 MB

Externals	Patches	Projects
mcl.jit.spihtaritenc.mxo	MclSPIHTarit_send.maxpat	Mcl_spihtaritenc
mcl.jit.spihtaritdec.mxo	MclSPIHTarit_recv.maxpat	Mcl_spihtaritdec
mcl.jit.spihtbinenc.mxo	MclSPIHTbin_send.maxpat	Mcl_spihtbinenc
mcl.jit.spihtbindec.mxo	MclSPIHTbin_recv.maxpat	Mcl_spihtbindec
mcl.jit.spihtROIaritenc.mxo	spihtarit_enc_dectest.maxpat	Mcl_spihtROIaritenc
mcl.jit.spihtROIaritdec.mxo	spihtbin_enc_dectest.maxpat	Mcl_spihtROIaritdec
mcl.jit.spihtROIbinenc.mxo	ROI_arit.maxpat	Mcl_spihtROIbinenc
mcl.jit.spihtROIbindec.mxo	ROI_bin.maxpat	Mcl_spihtROIbindec
mcl.jit.motion.mxo	Rate Control.maxpat	Mcl_motion

• mcl.jit_Windows 174.5 MB

Externals	Patches	Projects
mcl.jit.spihtaritenc.mxe	MclSPIHTarit_send.maxpat	Mcl_spihtaritenc
mcl.jit.spihtaritdec.mxe	MclSPIHTarit_recv.maxpat	Mcl_spihtaritdec
mcl.jit.spihtbinenc.mxe	MclSPIHTbin_send.maxpat	Mcl_spihtbinenc
mcl.jit.spihtbindec.mxe	MclSPIHTbin_recv.maxpat	Mcl_spihtbindec
mcl.jit.spihtROIaritenc.mxe	spihtarit_enc_dectest.maxpat	Mcl_spihtROIaritenc
mcl.jit.spihtROIaritdec.mxe	spihtbin_enc_dectest.maxpat	Mcl_spihtROIaritdec
mcl.jit.spihtROIbinenc.mxe	ROI_arit.maxpat	Mcl_spihtROIbinenc
mcl.jit.spihtROIbindec.mxe	ROI_bin.maxpat	Mcl_spihtROIbindec
mcl.jit.motion.mxe	Rate Control.maxpat	Mcl_motion