Using OpenGL for Video Streaming

by

Patrick Gilhuly

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Technical Report Number CS-2001-10

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The process of streaming digital video is difficult due to the large amount of data inherent. Previously, without specialized hardware, video output has been limited to low resolution or slow playback, as measured by frames per second.

Many of the functions of a video decoder can be mapped to the capabilities of a graphics card. However, for the graphics work involved with digital video a graphics card is required while a video decoder card is not. This thesis proposes a method to use common graphics cards, through the OpenGL API, for the task of video streaming.

Acknowledgements

I would like to thank Stephen Mann for guiding me throughout my Master's degree. I would like to thank Michael McCool and Gladimir Baranoski for agreeing to read my thesis. The comments and recommendations they provided were all helpful.

I would like to thank the people of the Computer Graphics Lab, who offered relief from the daily grind and help when I was stuck. Specifically I would like to thank Josée Lajoie, who created two animations that I used, and the team behind the Geromino animation.

This thesis was partially funded by the University of Waterloo Graduate Scholarship and a scholarship from ICR.

Trademarks

OpenGL is the trademark of sgi.

Onyx is the trademark of sgi.

Sirius Video is the trademark of sgi.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer animation has its origins in the early 1960s. One of the first applications of computer animation was a film made by Edward Zajac in 1961 [Auz94]. The film was used to display the results of his research regarding whether a satellite could be stabilized with one face constantly towards the Earth while in orbit. Computer animation is still used for simulations, but it is also presently used for advertising, entertainment and education.

The equipment for displaying animations varies greatly. If you are trying to maximize the number of people to whom you can show your work you quickly discover a conflict. Digital video viewers desire higher resolution formats because they are more visually appealing or, for simulations and medical applications, more informative. However, high resolution requires more computer resources and not all computers have sufficient resources. I am interested in the problem of streaming digital video and making it displayable on a wider set of computers.

The resources required to play a digital video depend on more than just resolution, they also depend on whether the video sequence has been compressed and what kind of compression has been used. Uncompressed digital video consists of a sequence of images in raw pixel data and requires large bandwidth and storage space. For local viewing, you can resolve the bandwidth requirement with specialized hardware such as a striped array of disks. Typically the bandwidth requirements of uncompressed video are too great to stream across a network. Uncompressed

digital video has the notable advantage that it does not introduce artifacts into the animation. This is critical when displaying a simulation since artifacts are equivalent to misleading or corrupted data. However, if you do not have sufficient bandwidth or storage space you will have to use compression. Compressed digital video does not require as much bandwidth, but requires greater processing since the stream needs to be decompressed during playback. This additional processing may be performed either on specialized hardware or, if you have a fast CPU, in software. Compressed digital video can introduce artifacts, especially as you raise the compression rate.

Over the past few years use of compressed video, in the form of Digital Versatile Discs (DVD), has become widespread. It is estimated that there are currently at least seven million installed DVD players, not including DVD-ROMs installed on personal computers. However, DVD players are highly specialized, and are manufactured to decode only NTSC and PAL format video streams. This restricts the animation creator to those specific resolutions, preventing both non-standard resolutions and higher resolutions such as HDTV. If you wish to use a resolution besides that which NTSC or PAL allows, you are forced to use a software decoder. Decoding in software is often undesirable since video decompression is highly CPU intensive. While recent personal computers are generally capable of decoding an NTSC signal in software, higher resolution formats remain too processor expensive.

In this thesis I propose another solution for displaying video. I propose a compressed format that does not attempt to maximize compression ratio, but rather uses compression only to reduce the required bandwidth to what is commonly available from a single hard disk. Further, instead of requiring either a digital video card that performs hardware decompression or a quick CPU, I propose utilizing graphics hardware for decompressing video streams. Graphics hardware is a common resource, and with the advent of 3D games is becoming more common and powerful. One might even speculate that development of graphics cards will be more rapid than the development of hardware decoding solutions due to the broader market. Modern graphics cards do much more than translate a digital signal to analog to link a computer with its monitor. Graphics cards are capable of modifying convex polygons covering screen data at any pixel. However, they do not

have all the capabilities of a video decoder card. This thesis will investigate how graphics card capabilities can be applied to the problem of streaming arbitrary resolution video streams.

1.1 Overview

In the second chapter I will discuss uncompressed video. I will outline why it is unsuitable for most users and then describe my implementation of an uncompressed video player. In the third chapter I will describe video compression techniques. Along with the advantages I will discuss some of the shortcomings of compressed video. In the fourth chapter I will discuss the capabilities of OpenGL 1.1, highlighting features that can be used for compression. In the fifth chapter I will outline the decoder I wrote for streaming with an OpenGL compliant graphics card. In the sixth chapter I will describe the encoder that I wrote. Finally I will present my results and conclusions.

Chapter 2

Uncompressed Streaming: An Implementation

2.1 Introduction

Although my goal is to stream compressed digital video, as discussed in the previous chapter, there can be advantages to uncompressed video. If you have the resources, both in bandwidth and storage space, uncompressed video may be preferable to compressed video since you can be certain that you will not introduce artifacts into your video. An uncompressed video streamer is also simpler to implement. In this chapter I will describe the hardware and software resources that I applied to develop an uncompressed video streamer.

2.2 Hardware

The computer I developed the streamer for is an sgi Onyx dual processor (150 MHz R4400) with 128MB of RAM. The Onyx was purchased in 1993, and so in most regards it is out of date. The Onyx has a Sirius video board attached, which links the computer to a Sony Hi-8 video recorder. One of the capabilities of the Sirius video board is to convert between different video formats in

real-time. The video recorder is limited to CCIR601 525 timing. The Onyx also has four 2GB hard disks attached in two pairs to two SCSI controllers. The hard disks are third party, and not resold by sgi with sgi's modifications. However, the disks are FAST, WIDE SCSI, which allows a maximum data transfer of 20 MB/s per controller [Sch93], so I had hard upper limit of 40 MB/s.

I only store the viewable, or active, region of each frame which, as dictated by sgi software, is 646×486 pixels, when using square pixels. Since I was interested in streaming thirty RGB frames per second at that resolution, I required a bandwidth of approximately 27 MB/s, which was well within the maximum achievable transfer rate. When I tested the disks, by reading a 36 MB file, I discovered that their maximum sustained read rate was 22 MB/s. As a result I was forced to use a different colour coding scheme to reduce the required throughput. This decision is detailed below.

If the disk space is devoted solely to video, and the video sequence is stored as uncompressed RGB data for the active region of NTSC frames, I am able to store approximately 6 minutes of video. This is sufficient for streaming short animations such as those developed in the Computer Graphics Lab. In addition, longer animations can be created one sequence at a time, and then transferred onto film or video tape. Thus, 6 minutes would suffice, although the resolution is limited.

2.3 Software

Since I was working on an sgi I used many sgi specific software tools. sgi provides an interface to video devices, a set of extensions for real-time programs and support for logical volume management of hard disks.

sgi provides a programmer's interface to video devices called Video Library (VL). VL allows the user to write programs that set a source and destination for a video signal. When an application uses memory as either the source or destination, which is necessary if you are streaming from disk or saving a video signal to disk, VL provides and manages a ring buffer for fields. VL abstracts the video devices so that the programmer can ignore many details specific to an individual device.

sgi machines run a UNIX variant. As a result at any given time several processes may be attempting to run programs in a time sharing environment. This is unacceptable for a real-time program because a real-time program can not tolerate arbitrary length interrupts at arbitrary times. sgi's real-time extensions can prevent a program from being interrupted by another process.

sgi has produced a logical volume for hard disks named XLV. A logical volume provides an abstracted view of a set of hard disk partitions. Individual volumes may vary from a single partition of a disk to several complete hard disks, but the volume manager makes them look like a single partition to users. XLV supports striping, which places consecutive blocks on consecutive hard disks in the volume. Multiple disks are required to create a stripe, and XLV requires that the disks have the same size. XLV fails to make the volume appear to be a single partition if the user chooses to include a real-time sub-volume.

On an sgi, you must create a real-time sub-volume if you wish to read from or write to the disk uninterrupted at a high rate. To gain this effect, XLV tries to isolate the data. XLV achieves physical isolation by insisting that the partitions that make up a real-time sub-volume must all be complete hard disks. This physically separates the real-time data from the regular user data. The user can enhance this physical isolation by placing real-time disks on one or more SCSI buses that are separate from the disks of regular users. XLV creates logical isolation by requiring a call to *fcntl* to place a file on the real-time sub-volume. The *fcntl* call must be made during the creation of the file, so standard applications are unable to create real-time files. Isolation is useful because it circumvents the problem of other users making interfering disk accesses when you are attempting to use the disk. Unfortunately, the logical isolation has the minor drawback of making the real-time sub-volume seem to be a separate partition.

By choosing to use a real-time sub-volume I was forced to use Direct I/O. The user selects Direct I/O as an option when opening the file. When using Direct I/O, data read from the disk is copied directly to the user's buffer, bypassing the kernel buffer. Direct I/O restricts the user to read a multiple of disk blocks at a time so that a block does not need to be accessed twice.

2.4 Achieving Read Rate

My first objective was to achieve the fastest read rate possible. By creating an XLV logical volume with a striped real-time sub-volume across the four 2GB hard disks, I was able to attain a maximum sustained read rate of 22 MB/s. This rate is only possible by having two dedicated SCSI controllers, since a fast, wide SCSI bus can only transfer 20 MB/s. My testing revealed that the read rate was inconsistent between files, with a low read rate of 12 MB/s observed. One speculation is that the slower read rate was caused by the physical location of the blocks on the disk. This hypothesis proposes that the blocks closest to the hub need a longer read time due to the greater arc length that the disk head must travel. An alternate hypothesis proposes that unsuitable blocks are placed in the same file so that much of the read time is actually spent seeking the next block. I could not test these hypotheses because I do not have direct control over the placement of the data blocks onto the disk. Fortunately the read rate of a single file is consistent. If I discover that a file has a poor read rate I can make copies until I create a copy with a desirable read rate. This raises the question, what is the minimum acceptable read rate?

Since my maximum observed read rate was only 22 MB/s I decided to change my bit packing to reduce the required bandwidth. I switched from RGB 4:4:4 to YUV 4:2:2 (YVYU). The YVYU format represents colours as a luminance channel and two chrominance channels, and the 4:2:2 notation indicates that the chrominance channels are each sampled half as frequently as the luminance channel. Studies have shown that the human visual system is less responsive to high frequency information in chrominance channels than in luminance channels, which suggests that high frequency chrominance data can be discarded without perceptibly altering an image. Furthermore, this down sampling will occur anyway when the video is streamed to the video deck, due to the video deck's limitations. For every two pixels I use two luminance values and two shared chrominance values instead of six colour values. By storing the animation in YVYU I require 1/3 less space and a 1/3 slower read rate is sufficient, as compared to RGB. Instead of requiring 26.9 MB/s, I now require only 18.0 MB/s. This made video streaming feasible with the existing equipment.

2.5 Final Implementation Details

The uncompressed video streamer uses three processes. One process reads from the disks while a second process writes the data from memory into the VL ring buffer. The processes communicate with semaphores, and use a ring buffer in shared memory. The semaphores indicate how much space is available for the reader, and how much space the reader has filled and is now available for the writer. I do not read directly from the disk into the VL ring buffer because the frame size does not match a multiple of the real-time disk extent size, and so is not a legal read size.

The third process detects errors during the streaming process. This is only for diagnostic purposes due to the difficulty of correcting an exceptional event. For example, the most common error is a dropped frame due to reading too slowly from disk, which, as mentioned in Section 2.4, is caused by poor physical placement of the file. Dropping a frame is problematic, both because of the visual appearance and the technical difficulty of seeking forward when the offending read may have happened at an unknown time in the past, due to the buffering of frames both by my streamer and by VL, and the amount to seek forward is not a multiple of the real-time extent size. The solution I chose was to write the animation in short segments. For this implementation I saved two seconds of the animation to each file. If any of the files were suffering from a low read rate, I repeatedly copied the file until one of the copies had a sufficiently fast read rate.

2.6 Conclusion

Although I was able to write an uncompressed video streamer, I was approaching the limits of the available hardware. For example, if I did not have access to the four hard disks and two dedicated SCSI buses for the real-time sub-volume, I would have been unable to achieve the necessary read rate. Graphics cards with enhanced capabilities are becoming more prevalent because they are required for modeling and animation and for 3D immersive, highly interactive environments. I am interested in showing that by modifying the compression technique to fit graphics hardware, a modern graphics card can be used to stream video. By using compression I hope to remove the need for a striped disk or real-time sub-volume, making video streaming possible on a broader

range of computers.

Chapter 3

Digital Video Compression Techniques

Compression is key to making digital video feasible for a broad audience. Compression reduces storage space requirements and bandwidth requirements during playback. However, when used inappropriately, compression can result in distracting artifacts in the video. Another drawback of compression is the processing time it requires during both the compression and decompression phases. This chapter will introduce compression and describe some of the more common compression techniques.

All compression techniques can be labeled as either lossless or lossy, and I will begin with a description of each to demonstrate the distinction. I will then discuss compression techniques that have been specifically designed for digital video. Finally I will outline the major features of MPEG, the most widely accepted digital video compression standard.

3.1 Lossless Compression

Compression takes an input message m and transforms it with a function or algorithm to create the encoded message m_e. By applying the inverse of the encoding method on m_e we obtain the

decoded message m_d. We call a compression technique lossless if m is identical to m_d for all messages m. The first thing to observe is that the encoding function for lossless compression techniques is one to one because the function has an exact inverse. Secondly, we can observe that a lossless compression technique will not always reduce the number of bits required to represent a message m.

Lemma 3.1 *Lossless compression techniques can not be guaranteed to compress a message without some prior knowledge of the message.*

Proof: Consider a lossless compression technique that uses function f to encode messages. We know that f is one to one. Consider all messages m with length n bits. There are 2^n such messages. The number of messages with fewer bits is $\sum_{i=1}^{n-1} 2^i = 2^n - 2$. From the pigeon-hole principle we know that there can not be a one to one function between these two sets. Thus no lossless compression technique can guarantee to compress all messages without knowledge of the message content.

This lemma confirms common expectations. Clearly one can not repeatedly compress a message and constantly make it smaller. If this were possible you would be able to reduce any message to 1 bit and then reconstruct the original message. You should not infer from this lemma that lossless compression is impractical. Typically we will use a specific compression technique for a specific application. In that circumstance we will have prior information about the message structure or content that will make compression possible. As examples, I will present the Huffman coding and run length encoding schemes, both of which are used for video compression.

3.1.1 Huffman Coding

Consider the problem of trying to compress English text. You can expect that between two messages, despite obvious differences such as length and content, the frequency of each letter will be remarkably similar. For example, ‘e’ is the most commonly used letter in the English language and you can expect that approximately one eighth of all letters will be ‘e’.

One approach to compression is to assign shorter codes to letters that are more common and longer codes to less common letters. This technique is known is variable length coding (VLC). To use VLC, you must have a finite alphabet $\Sigma = a_1, a_2, \ldots, a_n$, with known symbol probabilities $p_1, p_2, \ldots, p_n$. Among VLCs that assign each symbol a unique encoding, Huffman's algorithm produces an optimal variable length code with respect to the expected length of an encoded message.

3.1.2 Run Length Encoding

Huffman coding is poorly suited for image data. Although you could get general probabilities for various pixel intensities by sampling a variety of images, the compression rate would not be very high and the algorithm would not be as effective for all images. However, there are generalizations one can make about images. Images often have long runs of the same colour due to objects having a consistent colour. Images also have runs where the colour is inconsistent. These runs usually occur at edges. Run length encoding (RLE) takes advantage of this by encoding pixels as runs. The first byte sent indicates the type of run (one bit) and the length of the run. In the event of a run of a single colour the following bytes record the colour. Otherwise the following bytes are the colours of the pixels in the run. Typically the colour channels are treated separately because the colour channels vary independently. This technique can be taken a step further to a bit oriented run length encoding where each bitplane of each colour is treated separately. Although the compression ratios will vary from image to image, you can expect a compression ratio of 2:1 for byte oriented run length encoding.

3.2 Lossy Compression

Unlike lossless compression, which relies on a probabilistic analysis of the expected content or structure of a message, lossy techniques use expectations of what an observer will notice or find objectionable. As before, consider an input message m which has been encoded to m_e and subsequently decoded to m_d. If m_d is not identical to m then we classify the compression method

as lossy since some of the original data has been corrupted and hence lost. Lossy compression techniques are designed to minimize noticeable error while achieving a desired compression rate. The simplest and most widely used forms of lossy compression are sub-sampling and quantization.

3.2.1 Subsampling

We can consider an image to be an array of samples of a function. The value at each pixel is determined by the underlying scene. As with other functions it may be possible to recreate the original function, or a reasonable facsimile, with a subset of the original samples. At the simplest level, to reproduce a line you require only two samples, and additional samples do not add further information. To subsample a colour image you store less than the standard three colour values per pixel.

One method of subsampling an image is to determine the colour once for a block of neighbouring pixels. For example you might give every pair of pixels the same colour, or even every two by two block of pixels. By subsampling you can reduce the amount of data used to represent an image. The two previous examples would compress an image by a factor of 2:1 and 4:1 respectively. However, if you subsample an image across all colour channels you will noticeably diminish the quality of the image. Under severe subsampling the image will look blocky, gradients between two colours will have noticeable steps, and lines and curves will look jagged. However, subsampling can be used with little or no noticeable effect if we apply knowledge of the human visual system.

The human eye detects detail only at its centre. If you could successfully predict where a viewer would be looking in a video frame, you could maintain detail in that area while reducing the detail in the remainder of the frame. Unfortunately a viewer's point of focus during a video playback will change from time to time, depending on numerous variables independent of the video, such as the viewer's current interests. However, the eye does have other properties that we can exploit to increase compression rates.

This centre of the eye, known as the fovea, detects detail because it is densely packed with cones. There are three types of cones that detect light at different frequencies. Each type of cone responds to a range of frequencies of light, but one type responds strongest to blue light,

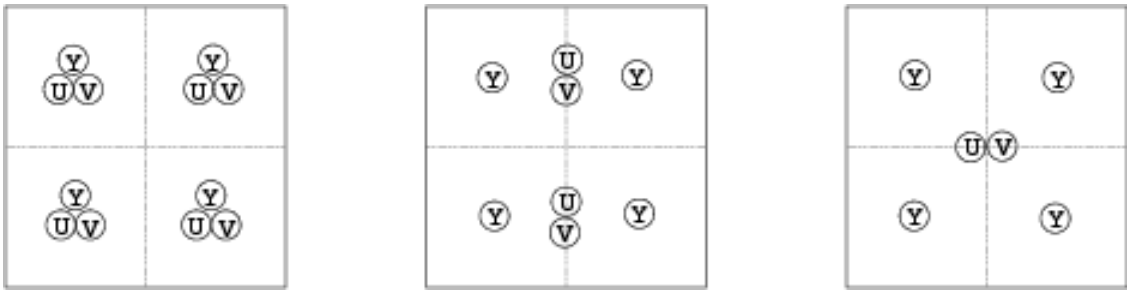

Figure 3.1: Logical sampling locations for YUV, YUV 4:2:2 and YUV 4:2:0.

another to green light and the last type of cone responds to red light. We can create the same response in the eye to a variety of colours by combining these three base colours. Computer displays work in this manner, combining different intensities of red, green and blue at each pixel to create the appearance of millions of colours. We believe that the eye will use any of its cones to measure the brightness of an image at a point, but requires a cone cell of each of the three varieties to determine a colour [Wat95]. From this we infer that the eye has greater resolution when determining differences in brightness than when determining differences in colours. This premise in turn suggests a sub-sampling method. First record each colour as a luminance value and two chroma channels instead of with a red, green and blue value. These representations are equivalent, but by separating the brightness from the colour we can subsample the image in its colour channels without creating noticeable artifacts. YUV is one common alternative representation to RGB. The Y channel is the luminance, or brightness, of the pixel while the U and V channels represent the colour. The two most common sub-sampled formats of YUV are YUV 4:2:2, which is sub-sampled in the two colour channels in the horizontal dimension 2:1, and YUV 4:2:0, which is sub-sampled 2:1 in both the horizontal and vertical dimensions. If we assume one byte per colour channel, the standard RGB representation uses twelve bytes to represent four pixels, while YUV 4:2:2 uses eight bytes and YUV 4:2:0 uses six bytes. A comparison of the two subsampled formats with straight YUV is shown in Figure 3.1 .

3.2.2 Quantization

While sub-sampling takes several values and replaces them with one representative value, quantization compresses by reducing the number of bits used to store a value. With image data, the values stored are the intensity for each of the colour channels at each pixel. Quantization is based on the belief that the final bit or two bits do not noticeably affect the intensity of a single pixel. If you quantize an eight bit colour channel by discarding the low two bits you will compress the image by 25 per cent. Quantization is not always done by dividing by powers of two; when combined with a bit stream coding method, such as Huffman coding, you can divide your input by any value you choose.

Unfortunately, while quantization may not be noticeable at the pixel level, it does introduce artifacts into the image. The most noticeable artifact created by quantization is called banding. Smooth transitions between two colours become a set of obvious bands of distinct colours because there are fewer levels for the intensity of each pixel and the difference between each level is more more significant and thus more noticeable.

3.3 Digital Video Compression Techniques

A video stream contains a large amount of data, but most of it is redundant. From one frame to the next it is possible for the camera to move, for objects in the scene to move or for the lighting to change. However, to maintain a sense of continuity, these changes have to be minor.

Although there are many compression utilities for digital video, they are all either image or model based. Image based methods treat a series of still frames as a series of related images. Model based methods treat frames, not as a series of flat images, but as a series of views of three dimensional objects.

3.3.1 Image Based Compression

Image based compression methods treat video as a sequence of pictures. Although it is understood that some three dimensional objects were used to create the images, no knowledge of these objects

is used. Image based compression uses both intra frame and inter frame methods.

Intra frame techniques use only the data from a single frame. They are equivalent to still frame compression techniques. Quantized DCT and run length encoding are standard methods, depending on whether lossy or lossless compression is desired. Inter frame techniques use data from more than one frame. Motion compensation is an example of inter frame compression. I will discuss the quantizated DCT and motion compensation below.

Quantized Discrete Cosine Transform

The quantized discrete cosine transform is the centre of the JPEG image compression standard. I will discuss the quantized discrete cosine transform in two parts. First I will describe the discrete cosine transform and then the quantization phase.

The discrete cosine transform (DCT) is used to translate from pixel colour values to the underlying frequency information. The DCT is generally used on fixed size sub-blocks of the image. JPEG chose 8×8 pixel sub-blocks as a compromise; a larger block offers larger potential for compression but a smaller block is faster to compute. The DCT can be written as

$$F(z_1, z_2) = \frac{1}{4}\, C(z_1)C(z_2) \sum_{x_1=0}^{7} \sum_{x_2=0}^{7} P(x_1, x_2) \cos\left(\frac{\pi(2x_1+1)z_1}{16}\right) \cos\left(\frac{\pi(2x_2+1)z_2}{16}\right),$$

where F is the frequency information, P is the original pixel information, $0 \leq z_1 \leq 7$, $0 \leq z_2 \leq 7$ and

$$C(z) = \begin{cases} 1/\sqrt{2} & \text{if } z = 0 \\ 1 & \text{otherwise.} \end{cases}$$

The DCT alone does not compress data. The reverse is true since the data can now have a wider range of values and it will no longer fit in a byte. For example, consider $F(0,0)$ for an 8×8 block with one byte per pixel. The original pixel value had a range of $[0, 255]$, but if all the pixels have the maximum value then $F(0,0) = 2040$. The original data could be stored in eight bits, but the transformed coefficient would require eleven bits to store without losing data.

Quantization is used at this point to compress this frequency data. A naive approach to

quantization is to divide each frequency coefficient by the same amount. This will result in compression, but it will also result in noticeable artifacts. A better approach is to quantize noticeable frequencies less than frequencies that the typical viewer will not notice. From psycho-visual studies we know that higher frequencies are typically less important that the lower frequencies. JPEG has a set table of quantization values for each of the 64 frequencies determined by experimentation [FGW97].

You can also compress the quantized data further using lossless compression techniques. We know that neighbouring blocks in an image are likely to have a similar colour. Since the first coefficient is the average of the block, you can use the value for a previous block to predict the first coefficient of the current block. Also, we know that most images have little high frequency information, so most of the high frequency coefficients will be zero. We can use run length encoding on the number of consecutive coefficients to further improve compression. Finally, by ordering the coefficients in a zig-zag pattern along the diagonal rows starting with the upper left entry and progressing to the lower right entry, instead of using a row or column orientation for the quantized coefficients, we place the high frequency coefficients together at the end. This results in fewer, longer runs of zeros, which makes the run length encoding more efficient.

Motion Compensation

Inter frame techniques use data from more than one frame. One approach is to use a past frame to predict the current frame, and all the methods I will discuss take this approach. An alternate method is to use a future frame to predict the current frame. If you use a future frame for prediction then the future frame must be encoded in the video stream out of chronological order so that it is available when you decode the earlier frames that depend on it. This requires the encoder to buffer all the frames between the first frame that will use a future frame and the frame being used for prediction. Although using a future frame for prediction is useful for compression, it is problematic to implement and adds little to this discussion.

A simple form of inter frame compression is to use the pixel value from the previous frame to predict the value of the pixel in the next frame. The error term is then transmitted so that the

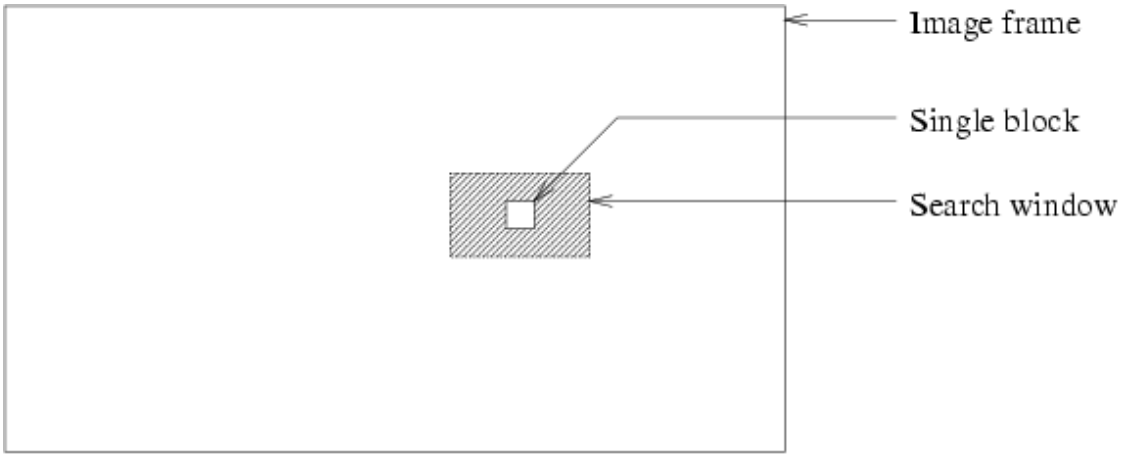

Figure 3.2: Search window for motion compensation blocks.

correct value can be reconstructed. You can not get large compression rates using this method because even when there is little difference between two frames many pixels can still have changed if only by a small amount.

A more sophisticated compression method is to attempt to determine the motion of objects between frames. Typically only translations are detected. Attempting to detect more complex movements, such as rotations, greatly increases processing time. For coding purposes each frame is divided into blocks. The block sizes vary for different applications, but 8×8 and 16×16 are typical sizes. For each block in a current frame, the encoder attempts to find the best match in the prediction frame. Since testing every block in the prediction frame against the current block is computationally expensive and because we assume that the motion between the prediction frame and the current frame will be relatively small, only a sub-window of the prediction frame is searched as shown in Figure 3.2. Since there are applications where it is too expensive to test every block in the search window against the current block, such as real-time encoding of video for television, heuristics have been developed to quickly find a near best block [FGW97]. Motion compensation methods then encode the error between the predicted block and the actual block. Motion compensation compresses video data more than basic intraframe techniques because it needs to code only a motion vector and a small error term instead of a complete block of image

data.

3.3.2 Model Based Compression

Although MPEG and similar compression methods are effective, they do not use the information available in a computer animation script and there is the expectation that greater compression ratios can be achieved for computer animations. Recently research has been applied to developing compression techniques that apply the information provided in an animation script. These methods may be thought of as model based because they utilize 3D modeling information.

Levoy presented a paper that suggested compressing an animation as a script and a stream of deltas [Lev95]. The script allows the streaming computer to create a low quality version of the animation, while the stream of deltas is the compressed per pixel difference between the low quality and high quality versions. The streaming computer can recreate the animation at a high quality level by adding the differences to the low quality animation.

More recently Cohen-Or et al. presented a paper to stream animations that are texture-intensive and have relatively few polygons [COMF99]. Cohen-Or et al. proposed using near views to construct intermediate frames. A near view is a frame from the animation, either from the past or future. Since near views are rendered images they can provide sophisticated lighting effects. Polygons in the current frame are textured by transforming the corresponding textured polygon in a near view to its current orientation. Cohen-Or et al. determines how much a polygon will be scaled from a near view to the current frame to decide when a texture needs to be updated. When a texture is stretched in either dimension by more than permitted by a quality setting, an update is transmitted for the near-view.

Currently model based compression has some weaknesses that need to be overcome. The examples presented by Cohen-Or et al. were small, only 180 pixels wide and 72 pixels high. Model based techniques require a script that computer animations have, which live video does not provide. The method Cohen-Or et al. presented for determining which texture should be used was based on avoiding stretching and not on determining correctness; if a specular highlight moves on a surface then the texture will be invalid. These are open research problems. Cohen-Or

et al. expect that with further research they will be able to overcome these limitations, but until then model based compression has limited usefulness.

3.4 MPEG Compression Standard

The Moving Pictures Expert Group (MPEG) has produced two standards for digital video and is working on a third standard. Each standard targets different throughput bit rates. I will discuss MPEG 1 because it is a simpler standard than MPEG 2 but has the same core compression techniques. Also I will refer to the standard as MPEG for brevity.

In an MPEG encoded video stream each frame is divided into 16×16 pixel macroblocks. The type of compression used on each macroblock depends on the type of frame. MPEG uses four different types of frames named *I*, *P*, *B*, and *D*. An *I* frame uses only intra frame compression techniques. Since MPEG uses YUV 4:2:0, each macroblock consists of four 8×8 pixel luminance blocks and one 8×8 block for each of the two chroma channels. Quantized DCT is used to compress each block individually. Further lossless coding is done, including predictive coding of the DC component for the blocks and Huffman coding of the quantized frequency coefficients. An *I* frame may be used by other frames for either forward or backward prediction. A *P* frame uses motion compensation to predict the contents of a macroblock. If motion compensation is used then the error term is encoded much like a regular macroblock. If no suitable motion vector is found for the macroblock, the macroblock is coded using the same method for macroblocks in *I* frames. A *P* frame can be used by other frames for forward or backward prediction. A *B* frame can use either forward prediction or backward prediction or the average of a macroblock from a previous frame and a macroblock from a future frame. A *B* frame can also use the same encoding method used by *I* frames. A *B* frame may not be used for prediction. A *D* frame only stores the DC component of each block. This yields a low quality version of the video stream that can be quickly displayed. Although *D* frames could be useful for searching through a video stream they are infrequently used.

An MPEG stream typically consists of a repeated pattern of of *I*, *P*, and *B* frames. One

possible stream is I, B, B, P, B, B, I,... with the second I frame indicating the position where the pattern begins to repeat. This stream would be streamed as I, P, B,... so that the P frame would be available for forward prediction when the first B frame was streamed. This requires additional sophistication in both the encoder and the decoder. Using a repeated pattern over a video stream can be inefficient since a scene cut invalidates the assumption that the previous frame is related to the current frame. Still, encoders often use a repeated pattern because it is easier to implement than a scene change detector. The MPEG standard requires an I frame every 132 frames to ensure that the encoder's internal representation of the video stream is equivalent to what the internal state of the decoder will be.

3.4.1 MPEG Artifacts

MPEG has been well studied and the artifacts that it creates are well known. Quantization is known to create contours and sudden colour changes because there are fewer possible levels. Since MPEG uses JPEG to encode base frames, at low bitrates too much high frequency information is removed and the result is a blurry image. Separate from that, the DCT can also introduce artifacts at edges where the colour changes radically because the DCT attempts to represent intensity levels of an image as a sum of cosine curves. The cosine curves can result in ghost images of edges that proceed and follow the actual edge. Finally, since MPEG operates independently on 8×8 pixel blocks, an MPEG encoded video can look blocky because neighbouring blocks that vary only slightly in colour might be assigned to visually distinguishable quantization levels.

Chapter 4

Features of OpenGL

OpenGL is a programming interface to graphics cards [WND97]. It is available on any current graphics card. OpenGL allows code to run on a variety of computers without the programmer knowing anything about the hardware inside. Naturally, how well the program runs depends on the quality of the OpenGL implementation and the hardware it is run on. The computer that I designed my video streamer for was an sgi Onyx with a VTX graphics board. I tested the speed of the OpenGL functions on the Onyx to determine which functions could be used for video decompression.

4.1 Colour Buffer

In a computer system, the graphics card controls the display. The graphics card maintains data for each pixel indicating what is currently visible on the monitor. This memory store is called the colour buffer, or sometimes the frame buffer. Each colour buffer element indicates how bright the screen should be lit at a specific pixel in each of the three colour channels. Negative values are not supported in the colour buffer to save memory since zero intensity is mapped to black. This makes some of the mathematics necessary for compression, such as change of basis, difficult to perform on the graphics card.

Pixel Map Size	Time (milliseconds)
8×8	0.072
16×16	0.086
32×32	0.131
64×64	0.255
256×256	2.086
512×256	5.412
512×512	10.250

Table 4.1: Time to display a pixel map at a random position.

The methods one uses to write data to the colour buffer depend on whether you are working in two or three dimensions. In three dimensions, the programmer makes changes to the contents of the frame buffer by indicating polygons in three dimensional space. Graphics cards have hardware to map polygons into the two dimensional space that the frame buffer represents, discarding any fragments of the polygon that are behind previously entered polygons. Since I am using a two dimensional approach to video compression I have disabled three dimensional processing. In two dimensional work you can still draw polygons, though everything is now at the same depth. You can also overwrite a section of the frame buffer with a pixel map. In Table 4.1, I display the time required to display a pixel map of various sizes on the target graphics card.

In this test I chose a random position for each draw location to demonstrate the flexibility of the pixel map drawing facility. In addition, I drew from a random, 4 byte aligned position within a 512×512 pixel map to reduce caching effects. If I had chosen a static position then it is likely that the smaller pixel maps would have been cached, leading to a deceptively fast time which would not be realizable in a more natural situation. The window that I was drawing into was also 512×512 pixels. As a result, the 512×512 draws have no random effects; there is only one sub-bitmap large enough, and only one place to draw the bitmap in the window. Finally, I performed this test with draw locations within the window fixed to an 8×8 grid and saw no performance improvement.

Pixel map drawing can not be used exclusively for video playback. Pixel maps exist in main memory, and if we load them directly from disk then we have an uncompressed video player, and we have not resolved the resource requirements. Decompressing a compressed video stream into

pixel maps and then displaying them will not work either, since the time to decompress the data stream is too long.

Fortunately, graphics cards have additional methods of altering the colour buffer information. In the following sections I will outline the more common functionality of graphics cards and how they can be applied to the task of video playback. I will also demonstrate whether they are suitable given the time constraints of video playback.

4.2 Double Buffering

Double buffering is the practice of using two buffers on the graphics card. While one buffer is used to update the monitor, changes can be performed on the second buffer. When the second buffer holds a completed frame suitable for viewing, the roles of the two buffers are switched. When using two buffers, the buffer that is displayed on the screen is called the front buffer and the buffer that is hidden is called the back buffer. Double buffering typically is used to reduce the amount of flickering in the screen.

When updating the screen the general process is to erase the objects that have moved and then redraw them in their new position. Since pixels on the screen are constantly being refreshed, when using a single colour buffer there is perceptible flicker because the user sees objects disappear and reappear in a new location. With double buffering, the intermediate step of erasing is not shown, and instead the user will only see the complete frames that he is supposed to see.

Double buffering does not have a noticeable time requirement, but it does double the amount of memory required for the colour buffer on the graphics card. Some graphics cards allow double buffering by reducing the number of bits they use for each colour channel in the frame buffer. As I have mentioned previously, reducing the number of bits used to represent a colour channel can lead to banding. Although both are noticeable artifacts, flickering is more distracting than the banding caused by a reduced number of bits for each colour.

You can also copy pixels between the two buffers. This feature can be used to write the contents of the front buffer into the back buffer so as to create a base image from which you can

Pixel Map Size	Time (milliseconds)
8×8	0.1033
16×16	0.1246
32×32	0.1879
64×64	0.3461
256×256	3.1111
512×256	5.7179
512×512	2.8457

Table 4.2: Time to copy a block of pixels between buffers at a random position.

add modifications to create the next displayed frame. In Table 4.2 I display the time required to copy different sized blocks of pixels from a random location to a random destination. The only copy without random location and destination was the copy of 512×512 pixel blocks which, given that the window was only 512×512 pixels, have only one possible source and one possible destination. One possible explanation for the exceptional performance for the 512×512 pixel blocks is that they were not copied from a random location to a random destination.

4.3 Blending

When drawing to the colour buffer the default action is to overwrite the values previously stored in pixels. However, there is also a blending mode, which uses the previous colour values when updating pixels in the colour buffer. In a blend, each of the current and the incoming colour values are independently scaled, and summed to yield the new pixel colour values. The most commonly used factor is the alpha value, either the alpha value stored at the pixel or the incoming alpha value. The alpha value is a fourth channel associated with each pixel, similar to the colour channels, except that it does not directly alter the appearance of a pixel. Other factors include 0, 1, the current colour values and the incoming colour values. Different factors can be applied to the current and incoming colour values.

If (r_n, g_n, b_n) is the set of colour values for a pixel in frame n and (r', g', b', a') is the set of incoming colour values, plus an alpha value, then the two blending possibilities that are most applicable to video streaming are

1. modulate the current colour with a drawn colour

$$\{r_n, g_n, b_n\} = (r_{n-1} \times r', g_{n-1} \times g', b_{n-1} \times b')$$

2. alpha blend the current colour with a drawn colour

$$\{r_n, g_n, b_n\} = (r_{n-1} \times a' + r', g_{n-1} \times a' + g', b_{n-1} \times a' + b')$$

   One would generate the drawn values r', g', b', and, a', given the current and previous frames as follows:

$$a' = min(r_n/r_{n-1}, g_n/g_{n-1}, b_n/b_{n-1}, 1)$$

$$c' = c_n - c_{n-1} \times a', c' \in [r', g', b'], c \in [r, g, b]$$

Note that each of the colour values is limited to $[0.0, 1.0]$. As a result, the first method has limited uses because each of the updated colour values must be no greater than the previous colour value, i.e., $r_n \leq r_{n-1}$. This technique may still be useful for generating fade out effects but it will not, in general, be useful for video decompression.

The second blending technique can be applied more often than the previous since it has a solution for any $\{r_n, g_n, b_n\}$ and $\{r_{n-1}, g_{n-1}, b_{n-1}\}$, again assuming all values are limited to the range $[0.0, 1.0]$. However, it has the disadvantage, compared to the previous blending technique, of requiring an additional value, the alpha value, in addition to the colour values.

In two dimensional drawing there are two drawing primitives that can be combined with blending: pixel maps and solid color polygons. Blending with a pixel map is not reasonable. Although it can produce the correct pixel values at each location, so can drawing a pixel map with blending disabled. Additionally, in some implementations blending may incur a small time penalty. However, blending with a fixed colour polygon might be used to simulate a variety of lighting changes. Since I was working with animations that had constant lighting, I did not

Accumulation Buffer Operation	Time (milliseconds)
Load buffer	6.64
Accumulate once	7.53
Return buffer	16.45

Table 4.3: Time required for basic accumulation buffer operations, using a 512×512 pixel buffer.

investigate this technique further.

4.4 Accumulation Buffer

The accumulation buffer is another facility for combining and blending results. The accumulation buffer holds memory for each pixel, but unlike the colour buffer, the accumulation buffer allows negative values. While the colour buffer clamps pixel values to $[0.0, 1.0]$, the accumulation buffer can hold at least the range $[-1.0, 1.0]$. The programmer can add or subtract the values of the colour buffer, scaled by any floating point number, from the values currently held in the accumulation buffer. In this manner it is also simple to take the average of several frames, which can be used to simulate depth of field and motion blur effects.

The accumulation buffer can be used for change of basis calculations because it supports positive and negative values. However, basis functions first need to be rendered into a colour buffer before they can be summed in the accumulation buffer. There are two common methods for handling negative values. The first method separates a function into its positive and negative values, and the second uses scaling and biasing of the values to reduce their range to $[0, 1]$.

A function f can be separated into its positive values, f^+, and its negative values, f^-, such that $f = f^+ - f^-$, $f^+ \geq 0$, $f^- \geq 0$ everywhere, and $\forall x$, either $f^+(x) = 0$ or $f^-(x) = 0$. I will represent function pairs with these properties as a pair such as $f = (f^+, f^-)$. Notice that these properties are preserved under multiplication. In the product,

$$(f^+, f^-) \times (g^+, g^-) = (f^+g^+ + f^-g^-, f^+g^- + f^-g^+)$$

only one of the two result components is non-zero because only one of the four terms is non-zero,

and the result components will also be non-negative since all the products consist of positive values. Calculations can be performed by drawing each term into the colour buffer. Addition of separated function pairs does not necessarily result in a functions pair that satisfies the property of only one function having a non zero value at any point. The accumulation buffer is required to normalize to function pairs after addition. For details see [Eve00].

The method of scaling and biasing maps a larger range of values into the range $[0, 1]$ which the colour buffer and texture memory can hold. For example, the cosine basis functions used in DCT calculations vary over the range $[-1, 1]$. By scaling the values by 0.5 and then biasing the values by 0.5 we shrink the range to $[0, 1]$, at the cost of some loss of precision. After evaluating a biased function in the colour buffer we need to remove the scale and subtract the bias. We call on accumulation buffer operations for this process.

If we consider a practical application, such as implementing an inverse DCT, we will see that accumulation buffer, for the target hardware, is not suitable for video decompression. The finite inverse DCT, as used in JPEG, can be written as [Wal91]

$$P(x_1, x_2) = \frac{1}{4} \sum_{z_1=0}^{7} \sum_{z_2=0}^{7} C(z_1)C(z_2)F(z_1, z_2) \cos\left(\frac{\pi(2x_1 + 1)z_1}{16}\right) \cos\left(\frac{\pi(2x_2 + 1)z_2}{16}\right),$$

where F are the per-frequency coefficients, P is the original pixel information, $0 \leq z_1 \leq 7$, $0 \leq z_2 \leq 7$ and

$$C(z) = \begin{cases} 1/\sqrt{2} & \text{if } z = 0 \\ 1 & \text{otherwise.} \end{cases}$$

We wish to adjust this equation to take advantage of the graphics hardware. First, we will precompute the value of the product of $C(z_1), C(z_2)$ and the two cosine functions. I will call these combined basis functions B_{z_1,z_2}. As well, we should think of the inverse DCT as a sum of matrix products. Finally, we can fold the constant $1/4$ term into the DCT coefficient $F(z_1, z_2)$. This reduces the inverse DCT equation to

$$P = \sum_{z_1=0}^{7} \sum_{z_2=0}^{7} F(z_1, z_2)B_{z_1,z_2}.$$

Since we are using lossy compression, we do not want to sum all sixty-four basis functions. From [YL95], we see that by summing ten DCT components, using a fixed bit allocation algorithm, we can reduce noise in the final image to acceptable levels. To evaluate the inverse DCT using function separation, i.e., $f = f^+ - f^-$, would require forty draws to the colour buffer, one accumulation buffer load, thirty-nine accumulations into the accumulation buffer and one accumulation buffer return. In Table 4.3 I display the time required for each of the basic accumulation buffer operations. The time required by the accumulation buffer operations would be 316.76 milliseconds, which would not be a problem if performed over several frames. The draws to the colour buffer, however, would require at least 3.5 seconds. This approach is inadequate for base frame generation since I wish to produce a new base frame approximately once a second.

The scale and bias technique can also be applied to evaluating the inverse DCT, and it is slightly more efficient. The operation we wish to evaluate is

$$2\left(\sum_{z_1=0}^{7}\sum_{z_2=0}^{7} F(z_1,z_2)B'_{z_1,z_2}\right) - \sum_{z_1=0}^{7}\sum_{z_2=0}^{7} F(z_1,z2),$$

where $F(z_1, z2)$ is as above, $B' = \frac{1}{2}B + \frac{1}{2}$ and the final sum is the correction term to unbias the calculation. The biased basis functions could be loaded into texture memory and negative biased basis functions could be loaded as well for use with negative DCT coefficients. The individual products could be calculated in the colour buffer by drawing a flat shaded 16×16 pixel square with modulated texture blending enabled. The summation $\sum FB'$ could be calculated in the colour buffer by using blending, although that may lead to clamping errors if the sum exceeds 1. After calculation the summation could be loaded into the accumulation buffer under a scale of two to undo the effect of the previous scale of the basis functions. The second summation could be calculated on the CPU and drawn into the colour buffer by drawing flat shaded 8×8 pixel squares coloured with the sum $\sum F(z_1, z2)$. This sum would have to be drawn in two passes, one for blocks where the sum is positive, and one for when the sum is negative. The positive values would be subtracted from the values in the accumulation buffer by summing with a weight of -1 and the negative values would be added to the values in the accumulation buffer by summing

with a weight of 1. This would require one accumulation buffer load, two accumulations and one return for a total of 38.15 milliseconds. If clamping errors occur, then the summation would have to occur within the accumulation buffer, eleven accumulations would be required instead of two, and the total time used would be 105.92 milliseconds. In addition to the time used by the accumulation buffer, the inverse DCT would require ten full screen draws of the colour buffer. If we are using a 512×512 pixel frame then these draws will take at least 0.9 seconds due to the full screen bandwidth limitation. If texture mapping with modulation enabled is slower than regular texture mapping this operation will take even longer. Although the accumulation buffer itself is fairly quick, and the task could be spread over many frames, since JPEG decoding is only needed for base frames, the cost of the draws to the colour buffer would still be slightly too high. Generating a new base frame could be done every second on the graphics hardware, but it would interfere with generating update frames for the video stream.

4.5 Texture Mapping

Texture mapping is used in modeling to add detail to a scene. For example, modeling a brick wall by defining the exact geometry and colour at each point is impractical for an application that does anything else. However, instead of recreating the geometry of a brick wall, you can use texture mapping to paste an image of a brick on a single rectangle. This gives the appearance of a wall to the viewer without requiring millions of polygons.

A texture mapping unit stores images in its texture memory buffer. Each individual element of texture memory is called a texel, and they are the equivalent of pixels in the colour buffer. There are restrictions on the number of texels to each dimension due to hardware limitations. For example, the dimensions of a texture map are restricted to a power of 2, and the maximum size in each dimension is also restricted.

You invoke texture mapping by indicating a correspondence between the vertices of a polygon you are drawing and a location in texture space. Texture space exists in the continuous range from 0.0 to 1.0, and represents an abstraction from the actual memory. Textures may be stretched, if

Textured Block Size	Time (milliseconds)
8×8	0.006
16×16	0.009
32×32	0.028
64×64	0.101
256×256	1.505
512×256	2.978
512×512	5.893

Table 4.4: Time to display a texture mapped block.

one texel covers many pixels, or filtered, if several texels are in one pixel. The texture provided can modify the colour of the pixel through modulation or blending. Alternately, the texture may be used to replace the colour of the polygon.

The main reason for using texture memory is its speed. Comparing Table 4.4 with Table 4.1, we see that texture mapped polygons can be drawn faster than pixel maps of the same size. At the smaller block sizes, the time difference is particularly pronounced, and the difference is still greater than a factor of 2 for 64×64 pixel blocks.

Chapter 5

Decompressor

A video streamer must decode an input stream into a sequence of frames and display the frames at a fixed rate. The standard rate for film is 24 frames per second and for television in North America it is 30 frames per second. Due to the size of the data involved, buffering all the frames before displaying them is not practical. Thus the next frame must be decoded during the time the current frame is displayed. To reduce the running time, I used a simple design.

At its simplest, a video streamer needs to be able to draw base frames and update frames. A base frame is a frame that is independent from all other frames. Base frames are necessary as a starting point and to curtail error propagation. Since I was unable to devise a graphics hardware accelerated image decompression algorithm, due largely to the inaccesibility of signed arithmetic, I chose to use JPEG encoding for base frames. The JPEG images would be decoded in software and copied to the screen as pixel maps. Update frames take advantage of continuity between frames. They encode the motion that occurred between the current frame and the next frame as blocks that have changed position. In addition, update frames require a form of error correction for portions of the image that can not be predicted from the reference frame. For motion updates, I implemented motion blocks as texture mapped rectangles. For error correction I used detail blocks, small pixel maps that could be drawn over errors in the decoded frames. In the remainder of this chapter I will describe the video streamer in greater detail and I will discuss some test

cases I ran.

5.1 Implementation

I wrote a video streamer to demonstrate the possibility of using graphics hardware for video decompression. The video streamer has three components: a disk reader, a JPEG decompressor and a graphics command issuer. The three components were implemented as threads with shared memory.

The disk reader component reads the data stream from the disk and separates the data into two substreams: one stream for motion and detail blocks and the second stream for JPEG strip data. The disk reader uses semaphores to coordinate with both of the other two components, and to prevent itself from monopolizing the CPU.

The JPEG decompressor component uses a system library to decompress JPEG encoded data into raw image data. It reads from the JPEG strip stream and outputs into a raw image stream. The original images are separated into 32 pixel high strips so that JPEG decoding, which is a CPU intensive operation, is spread over the playback of many frames. The JPEG decompressor component uses semaphores to coordinate with the disk reader component, which provides it with JPEG data, and the graphics command issuer, which it provides with raw image data.

The graphics issuer component is the actual video streamer. It has two modes of frame creation: base image display and image update. During base image display, raw image data is taken from the raw image stream and drawn as a pixel map on the screen. During image update, data is taken from the stream containing motion and detail block information and interpreted as a set of motion compensation commands and a set of detail blocks.

I perform motion compensation by reading the previous frame into the texture buffer. Then each motion compensation command becomes a straight forward texture mapping draw.

The detail blocks are stored as coordinates to the bottom left corner for the position and raw image data for the pixel information. The detail blocks are drawn into the colour buffer as pixel maps.

For each frame in the playback I use an extension to OpenGL named GetVideoSync and provided by sgi that allows a program thread to sleep until the next electron gun retrace occurs. GetVideoSync keeps count of the number of retraces, n, and given a modulo m and a congruence class c, it allows a process to sleep until $n \equiv c \pmod{m}$. Since the electron gun retrace runs on a constant frequency, this extension allows the programmer to maintain a desired play rate for their video. For example, drawing a frame on every other screen retrace ($n \equiv 0 \pmod{2}$ or $n \equiv 1 \pmod{2}$) on a monitor running at 60 Hz would yield an animation running at 30 frames per second. Unfortunately, GetVideoSync is flawed since the retrace count will not report occasions when I exceed the time limit for drawing a frame. This problem was detected by using another timer; the second timer reported in some cases that I had taken twice as long as the number of retraces reported by GetVideoSync would suggest. Despite this failure, GetVideoSync helps to maintain a steady play rate, and to keep to a specified rate I limit the amount of work that the encoder can request per frame.

5.2 A Simple Test

I tested my decompressor with a simple video sequence. The video sequence features a bouncing ball traveling from left to right against a flat shaded background. The individual frames measured 512×256 pixels and the sequence consisted of 171 frames. The ball has a simple texture map, and is diffusely lit. The frames for this animation were generated using Maya software by Josée Lajoie of the Computer Graphics Lab.

I created the compressed video stream by locating the ball in the previous and current frame and writing a motion block to represent the ball's motion. Due to the simplicity of the scene (the ball was orange and the background was blue) it was easy to locate the ball. After the motion block was recorded and the previous frame was updated to appear as it would after the motion block was applied, there was still error. The pixels were then sorted by difference between the reproduced colour and the desired colour. I then covered the high error pixels and their neighbours with 8×8 pixel blocks taken from the original image. The resulting stream used 0.66 bits per

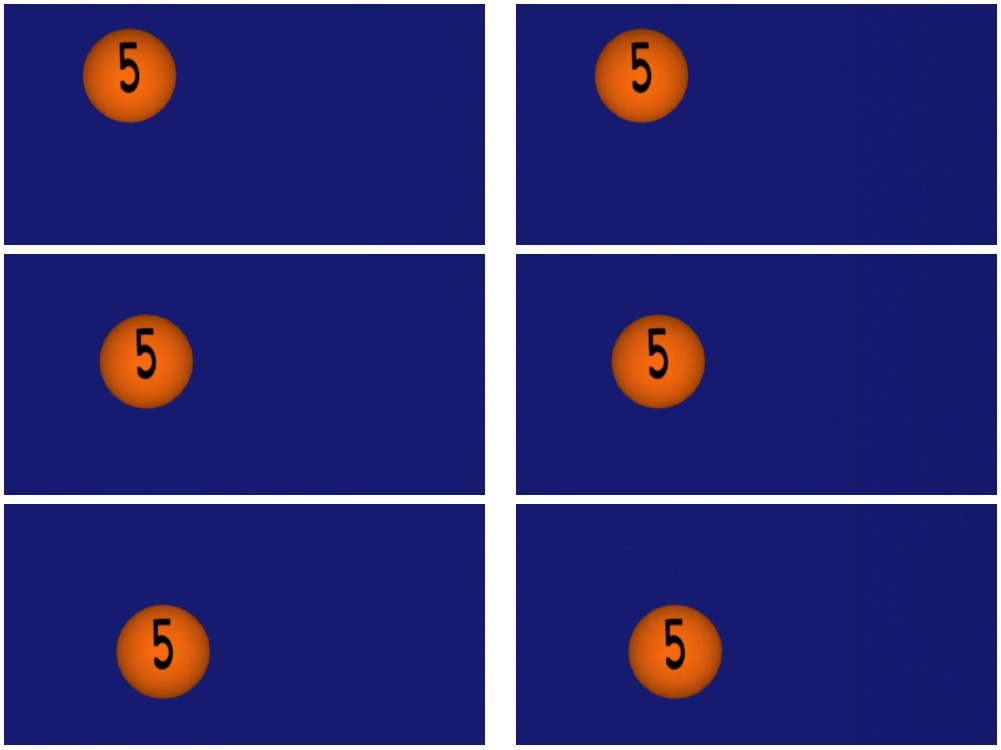

Figure 5.1: Sample images taken from the animation, with original frames on the left and reproduced frames on the right.

pixel. This is more than double the size required by MPEG, which only required 0.30 bits per pixel. However, during playback the motion of my decompressor was much smoother than that of the MPEG decoder.

Reporting the quality of an animation is difficult. The standard judgement, root mean square error, does not capture potentially objectionable artifacts. Qualitative judgements, while potentially more discriminating, are not objective. I have chosen to report both a qualitative and quantitative measurement.

Qualitatively, the decompressor displayed the video sequence well. The decompressor played at approximately 24.2 frames per second, where 24 frames per second was the desired rate. The visual quality was acceptable, although perhaps a little blurry. No other artifacts were noticed during the playback. See Figure 5.1 to compare sample original and reproduced frames.

To obtain the frames and perform a quantitative measurement I captured and saved frames from the colour buffer. Capturing frames alters the speed of the playback and may have had other unknown side effects on the video quality. Images saved from the colour buffer had an average mean square error of 6.05, and a high mean square error of 20.42. This is low, and should be taken to support the qualitative report that the video was visually acceptable.

This example is only a proof of concept to demonstrate the functionality of the decompressor. The compressor used for this video sequence is highly specialized and will not work for general video sequences. However, to prepare for displaying general video sequences, it is important to know how many block operations can be performed while maintaining a desired frame rate. Although I performed tests on individual functions earlier, it is important to perform a combined test of all the operations required for video streaming to determine whether there are any negative interactions between the various operations. Information on how many blocks the decoder can handle can be used to tune the encoder to output the desirable number of motion and detail blocks.

To further determine the limits of the graphics hardware, I altered the stream for this animation to specify the exact number of motion and detail blocks in each frame. Each run consisted of playing 171 frames modified to draw a specific number of motion and detail blocks. I varied the number of motion blocks from 5 to 200, although each motion block in a frame was identical. I also varied the number of detail blocks in each frame from 5 to 300. For frames where a larger number of detail blocks were provided than the number I wished to test, I discarded the extra blocks. For frames that provided fewer detail blocks than the number I wished to test, I used specified blocks multiple times. The timing results of the runs can be viewed in Figure 5.2, which also shows the contour curves on the plane below the graph.

There are two aspects to the results of the graph in Figure 5.2 that are worthy of note. Firstly, it is initially surprising to note that detail blocks have little effect on the time to display the video sequence, while motion blocks have a significant effect. This is surprising because motion blocks are drawn using texture mapping, which is faster than drawing pixmaps for blocks of equal size. By examining the relative sizes of the blocks the difference in impact on time becomes

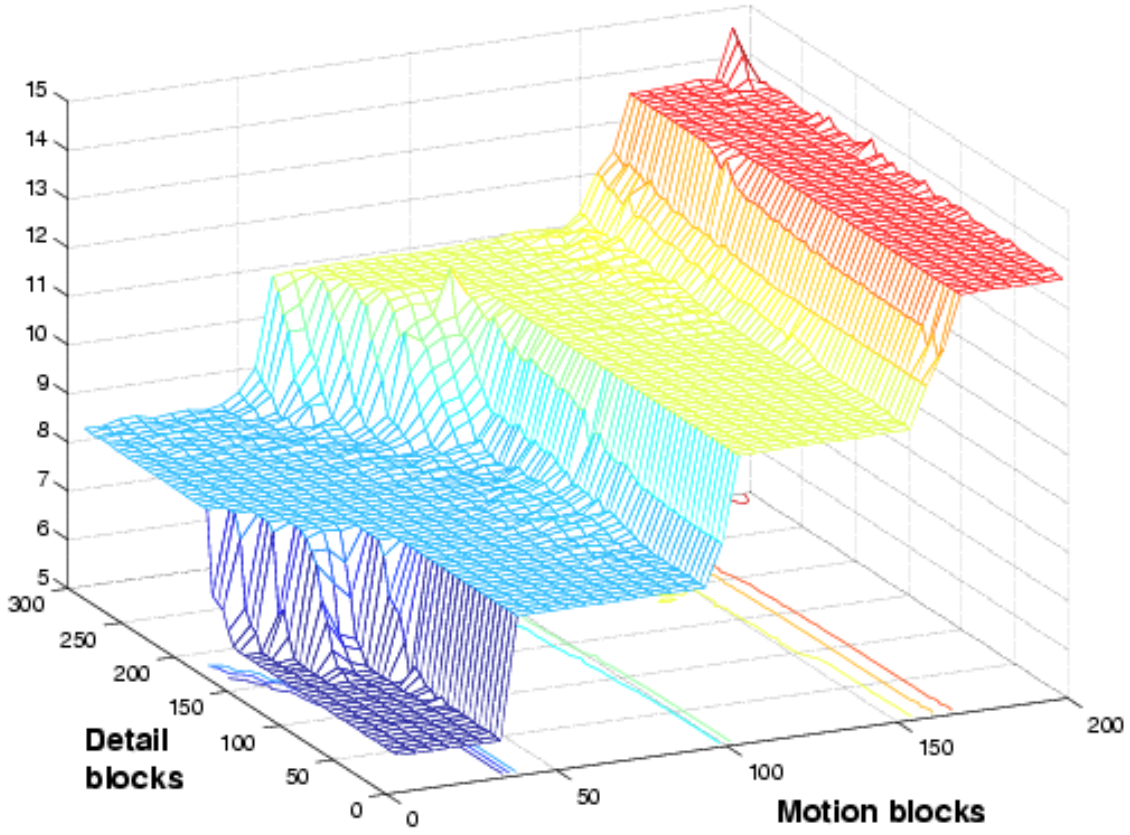

Figure 5.2: Test results for streamer with specified number of motion and detail blocks.

understandable. While the detail blocks are only 8×8 pixels, the motion blocks average 104×104 pixels. Considering the difference in size of the two block types, the difference in impact on playback speed seems less surprising. Secondly, there are four plateaus with sharp rises between them, instead of a gradually increasing slope. This is a side effect of the timing mechanism I use. The OpenGL extension GetVideoSync appears to drop vertical refresh events when the graphics card is under a heavy load. However, it is positive to note that GetVideoSync drops the same number of vertical refresh events when given a variety of different block counts. This consistency results in the plateaus of run time. Also, we can see from the graph that I can get 20 frames per second with a variety of motion and detail block combinations, specifically the combinations lying between the two contour curves that lie approximately at 40 motion and no detail blocks and 100 motion and no detail blocks. Although 20 frames per second is slower that the 24 frames per second that film uses, the motion in the displayed video will still appear acceptably smooth.

In the following chapter I will discuss my efforts to create a generalized video compressor that would produce an input stream for my decompressor.

Chapter 6

Implementation Notes

A digital video streamer must cope with change between frames, but not random change. Generally we expect most of the changes in a sequence of frames to be caused by motion; either motion of objects within the scene or camera motion. However motion compensation can not account for all the possible changes. In MPEG, for example, additional changes in the scene are encoded as error blocks. In my implementation I encapsulate changes between two frames with motion blocks and detail blocks. In the following sections I will outline the file format I use, named MPOG, to store video segments and the method I use to detect motion and detail blocks in the encoder.

6.1 MPOG file format

The MPOG file format has a simple layout, with three types of segments. The file is a byte stream instead of a bit stream for two reasons. Firstly, I wished to reduce the amount of processing required by the CPU. Secondly, I noticed that the compression rate gained by writing as a bit stream was low, well under a factor of 2 for most circumstances. The byte stream commences with a base image encoded in JPEG format. The base images are JPEG encoded and decoded by system libraries, so internally they will be bit streams. My encoding and decoding software does not require any knowledge of the internal structure of the JPEG data segments, and treats them

Image Strip Size	Time (seconds)
512×8	0.034
512×16	0.036
512×32	0.059
512×256	0.210
512×512	0.274

Table 6.1: Time to JPEG decode an image strip.

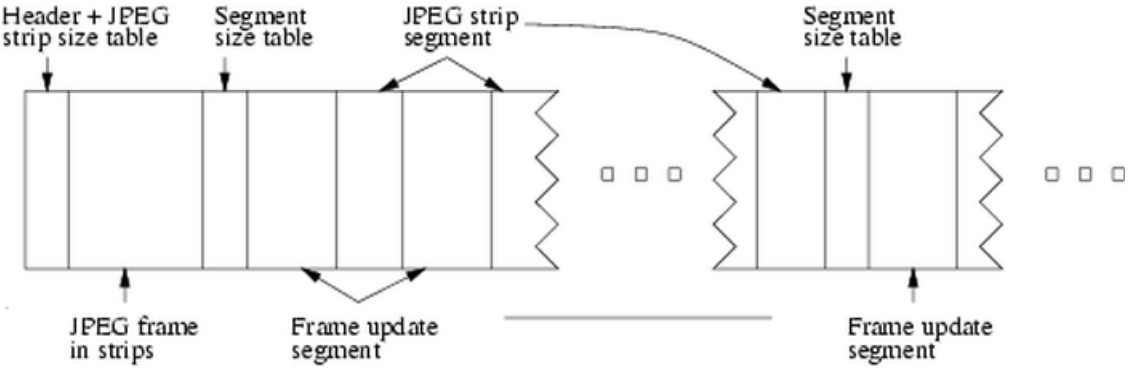

Figure 6.1: Block view of the file format.

as a part of the byte stream. Following the base image is a fixed size table that holds the sizes of the segments that follow the table. The following segments alternate between motion/detail block segments and JPEG strip segments.

To attempt load balancing, base images are encoded as horizontal strips of the image and not as an entire image. JPEG decoding of a 512×512 pixel image requires 0.274 seconds, as displayed in Table 6.1. By splitting the image into many strips I can decode a JPEG strip while playing a motion compensated frame during playback without delaying the video sequence. I currently use a strip height of 32 pixels to keep the number of frames between JPEG encoded frames low.

Due to the pattern of alternating interframe sequences and intraframe sequences, the number of frames between base images is fixed at *(height of image)/(strip height)*. By increasing the strip height I would increase the number of base images in the sequence and consequently might increase the quality of the video. Naturally, this assumes that the CPU could handle the increase in workload during playback due to the greater amount of JPEG decoding.

Having a fixed number of frames between each base image is not desirable. Scene changes, which invalidate the assumption of motion compensation, do not occur on a fixed period. While it would be possible to alter the file format so that there is a range of values for the number of frames in between each base frame, detecting the end of a sequence is a difficult problem and would add greatly to encoding time.

The motion block and detail block information for updating a frame are included in the same segment, called an update segment. The size for each segment is stored as 2 bytes in the segment size table, so the segment is restricted in size to 64 kB. The first two bytes for an update segment holds the number of motion blocks in the segment. Each motion block requires 8 bytes: 4 bytes for the x and y positions, 2 bytes for the x and y motion vectors, and 2 bytes for the x and y sizes of the motion block. Two bytes are more than sufficient for each position value since I restrict image size to be less than 512 pixels in both dimensions. One byte for each motion vector dimension allows a range of $[-128, ..., 127]$, which is much larger than the search window in which I search for the motion blocks. Finally, in my block detection algorithm in the encoder I limit block sizes in both dimensions to no more than 255 pixels, so a single byte is sufficient for block size.

The two bytes following the motion block information is the number of detail blocks. Detail blocks are fixed in size at 8×8 pixels. Since I could not devise a compression method for detail blocks that exploited the target graphics board, detail blocks are stored as raw data. Detail blocks also use 4 bytes for x and y position, for a total size of 196 bytes each. As a consequence, to reach a compression rate of 10:1, when compressing a video sequence with dimensions of 512×512 pixels, the number of detail blocks must be limited to fewer than 400 detail blocks. From the testing in the previous chapter, we know that 400 detail blocks is far too many to maintain a sufficiently fast playback rate. Using the data from the tests outlined in that chapter I have chosen to limit the number of detail blocks used in any frame to 200.

6.2 The Encoder

The encoder performs three main tasks: it separates base images into strips and performs JPEG encoding on each strip; it detects motion blocks and it places detail blocks to cover errors in the recreated frame. The first task is not interesting because I use a system library to perform the encoding and each strip has a fixed width. In the following sections I will discuss the algorithms I use for motion detection and detail block placement.

6.2.1 Motion Block Detection

The first phase of encoding an intermediate frame is to detect motion within the image. Motion detection is a difficult task, and there has been a great deal of research on this problem.

Although objects in motion might be best represented by a polygon of varying and arbitrary degree, I decided to limit my search to rectangles for two reasons. First, it reduces the complexity of the search algorithm, and consequently reduces encoding time. Secondly, I wanted to ensure that I would get the same performance on any graphics card. The pixels chosen to lie under a non-horizontal and non-vertical line may differ between hardware implementations. These errors are admittably small, but they may compound over several frames into a noticeable artifact.

When searching for rectangles I use the common assumptions that motion is generally linear, motion will be small from one frame to the next, and lighting changes infrequently. As a result I expect that if a group of pixels change position between two frames, the pixels will retain their position relative to each other, they will be close to their former positions and the colour value of each pixel will be similar to its value in the previous location in the previous frame.

The assumption that motion can be approximated as linear allows me to ignore the fact that objects are moving under a perspective transformation. For example, if an object moves into the background it will become smaller. However, the object will not move into the background so quickly that it becomes noticeably smaller from one frame to the next. By ignoring the effects perspective have on motion I increase the speed of motion detection.

The assumption that motion is small between two frames allows me to reduce the area that I search for motion. This area is known as the search window (see Figure 3.2) and it consists of all

the potential replacements for a pixel. Naturally there is a trade off between search space size and the quality of matches. Although it is possible that a block outside the search window is the best match, by limiting the scope of the search I increase the speed of motion detection, and reduce memory requirements for the search.

Having described my basic assumptions, I can describe my implementation for locating motion blocks. I use three image buffers to hold the image information I need. The first buffer holds the current frame. This is the frame that I am trying to recreate. The second buffer holds the reference frame. Motion blocks come from blocks in the reference frame which, under a translation, closely match blocks in the current frame. In my implementation the reference frame is also the previously generated frame. The third buffer holds the reconstructed frame. During encoding the reconstructed frame represents the image that will be created by the decoder, given the motion and detail blocks included to that moment.

While I do use a search window for motion blocks, I view it as a set of motion vectors instead of a window around each pixel. Each motion vector represents a translation of the reference frame. Associated with each motion vector is an array of structures, with the array size equal to the number of pixels in a frame. The structures hold information on a per pixel basis. The structure indicates whether this pixel, translated by the motion vector, is a suitable replacement for the same pixel in the current frame; the amount of error reduced by using this pixel and motion vector; the size of the best motion block anchored at this pixel; and the average amount of error reduced by that motion block.

There are many metrics for determining whether the pixel is a suitable replacement for a pixel in the current frame. I decide this by taking the absolute difference of each colour channel and transforming into luminance and two chrominance channels. Since the chrominance values naturally fall in the range $[-128, \ldots, 128]$, I take the absolute value, thus giving the luminance equal weight with the two chrominance channels.

The amount of error a pixel fixes is determined as the amount of error currently at that pixel, less the amount of error introduces by the pixel under the motion transform.

Deciding the best motion block anchored at a pixel with a given motion vector is the key

operation. I use the term anchored to indicate that the block has a fixed location. In this case the block is fixed with its upper right corner at the pixel specified. Finding a motion block requires compromises. Although a perfect match would be ideal, most often blocks that are perfect matches are small, and so carry greater expense in time per pixel fixed to recreate as compared to a larger block. Since detail blocks will be added subsequently to cover errors, it is not necessary to have perfect motion blocks. However, a block that covers pixels that do not need to be corrected or a block that contains many pixels in error is also undesirable.

The measure of quality for a block is the amount of error that it corrects less a small penalty per pixel. The penalty can be considered to compensate for the extra time required to draw extra pixels. The penalty is meant to prevent rows or columns of pixels on the edge of a block that do not add any value. All blocks anchored at the specified pixel are tested to find the best block, until the block becomes unsuitable to continue searching.

To permit imperfect blocks, and to limit how inaccurate they can be, I keep track of holes within the block. Holes are a prediction of where a detail block should be placed, due to the pixels being unsuitable replacement candidates. I limit the number of holes per block and the number of holes introduced per line. All incorrect pixels must be placed within a hole. If no hole can accommodate the incorrect pixel and the block has reached its limit for number of holes, then the search for best block at that pixel concludes. Similarly once two holes are allocated on a line, if an additional incorrect pixel is found that can not be added to a hole, then the maximum width for blocks anchored at this pixel is reduced to the number of pixels from the anchor pixel up to the first incorrect pixel that could not be added to a hole.

Calculating the best block for all pixels and all motion vectors is computationally expensive. I cull pixels that are unlikely to have a good associated best block to reduce the time required to compress a video sequence. For example, I cull pixels that are within the minimum block size of the bottom edge of the frame because any best block found for that pixel will be too small to be accepted. I delay evaluation of the best block until necessary. The best block for a pixel will be evaluated only when the block is a potential candidate for best available block. If a pixel is in an area of low or no error, then its best block will not be evaluated. Also, since there is a minimum

size to the block dimensions, pixels within that minimum distance from the bottom or left border will not have their best block evaluated.

The final value I store for each pixel and each motion vector is the average motion block value. I store the average per pixel improvement to reduce the memory required. When comparing two blocks I multiply the average block value by the block area to obtain the true block value.

As I mentioned previously, the evaluation of best block for a pixel is delayed until necessary. I order the pixels to evaluate best blocks by sorting the pixels in decreasing error between the current frame and the reference frame. Note that since I am sorting within a very limited range I can use a bucket sort variant to sort pixel error levels in linear time [CLR90]. For each motion vector I determine the best block at the specified block and at all pixels within the maximum block size to the right or above the specified pixel. To be considered the motion block must contain the specified pixel. From all these pixels I choose the block with the best value. The error of all pixels covered by the chosen motion block is reset to zero since another motion block is not required for those pixels, and furthermore an additional motion block may well be undesirable.

Motion block overlap in the current frame is undesirable since it may result in an unexpected colour value in the overlap region if the pixel writes do not occur in the order that you expect. To avoid motion block overlap in the recreated frame I store a map of the pixels that have been covered by a motion block already. When adding a new motion block I take the largest sub-block that does not overlap any of the already covered pixels. Best blocks are not culled before selection due to the computational expense.

Coding Improvement to Brute Force Approach

The brute force approach mentioned above has several significant drawbacks. The brute force approach requires too much memory, does not produce the best motion blocks and it takes too long to run.

First, the amount of memory required for to hold the data structures is vast. If we consider all motion vectors (x_1, x_2) such that $\mid x_i \mid \leq 8$, we have 289 motion vectors to consider. If we store one byte per pixel each for suitability, error reduced by the pixel, average error reduced by the

best block and two bytes per pixel for the best block size, then for a 512×256 pixel frame, we require 180 MB. This value is excessive, and it grows proportionally with the frame area. Clearly, less data should be stored, so some thought is required regarding what data may be discarded with the least impact on the encoder.

This brings us to the second problem, that often the motion blocks selected are not ideal. The effectiveness of the block rating scheme is at least partially responsible. However, another difficulty is that best blocks may be selected on erroneous data. When searching for a motion block to cover a specific error pixel, the encoder searches for the best block anchored at pixels above and to the right of the error pixel. The value for best block is set in these pixel locations, and not updated when new motion blocks are selected. A newly selected motion block is cropped against the pixels that have already been covered by a previously selected motion block. As a result, the resultant motion block may not reduce error as much as it claimed, and further it may not be the best motion block possible. There are two solutions to this problem. In the first solution, any evaluated best block that overlaps pixels covered by a selected motion block must be discarded and re-evaluated. This will only exacerbate the problem of a slow runtime. The second solution is to not store best block data, and to always re-evaluate. Although this second solution would greatly reduce the amount of memory required, it promises to increase the run time even more. Re-evaluation, in either form, has the benefit that it can eliminate the need for cropping motion blocks to prevent motion block overlap. Instead, pixels which have been chosen for a motion block can be marked unusable in future calculations, which will prevent motion block overlap.

The third problem with the brute force algorithm is its poor run time. It can take several hours to evaluate the motion blocks for a single 512×480 frame, and that is without re-evaluating data. When searching for a motion block to cover a pixel in error, the brute force algorithm tests at each pixel within the maximum block size to the right and above the error pixel. The algorithm culls the search only when it is impossible to find a motion block that will cover the error pixel. Instead, we can test on a evenly spaced grid, and then focus further attention on the most promising candidate. This method is modelled after the three step search used for finding

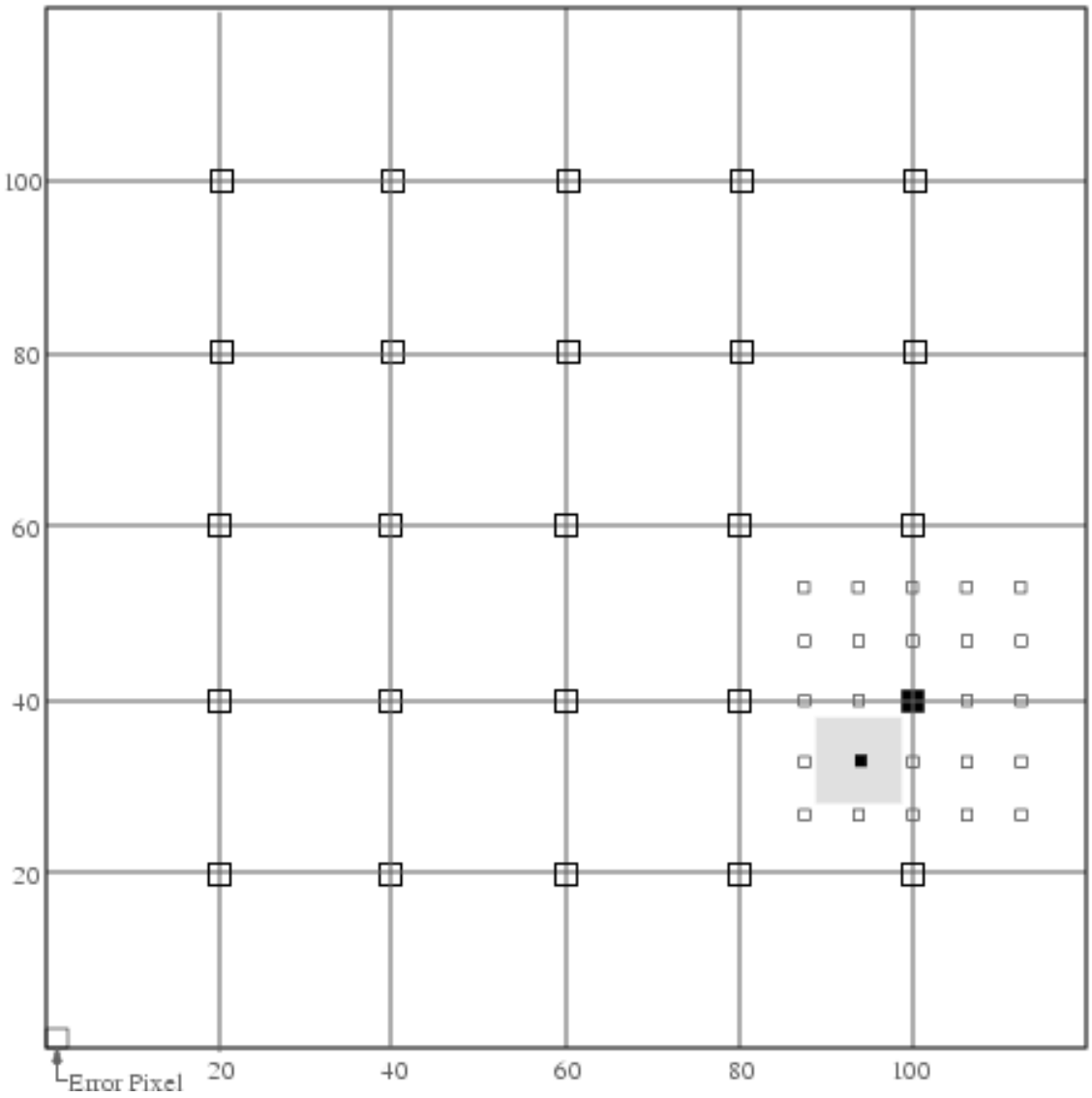

Figure 6.2: Hierarchical sampling grid to detect motion block. The first two levels are displayed, with the pixels chosen coloured black. Approximate spacing of the first level of samples is indicated on the two axes in pixels.

best motion vectors in MPEG encoding [Tek95]. See Figure 6.2 for a demonstration of the first few levels of selected pixels in a hierarchical evaluation.

In addition to the brute force encoder, I encoded a hierarchical encoder. The hierarchical encoder does not store information on the best block at each pixel, which greatly reduces its memory requirements. It also uses the hierarchical approach to deciding which pixels should be tested to find their best block, as outlined above. In my implementation I sample on a 5×5 grid evenly spaced among the possible anchor points for the best block. There can be up to four levels of evaluation and up to 97 pixels tested for their best block. This is much lower than the upper bound of the brute force algorithm which may test as many as 16384 pixels. Since not every pixel is tested for its best block it is possible that the best block that is found is a local error minimizer and not a global error minimizer. However, the hierarchical approach promises to be much faster than the brute force method, and it is likely that a local minimum will suffice. I compare the brute force and hierarchical methods in Chapter 7.

6.2.2 Detail Block Location

Motion blocks can not perfectly reproduce the current frame because motion is not the sole change between frames. Lighting could change, for example, or the camera could move non-linearly. Even when motion is the sole effect, it can create errors that are impossible to correct with motion compensation. When an object moves, it reveals the scene behind it. If nothing in the previous frame matches the revealed section of the screen, motion compensation can not correct the error. MPEG allows each block to use JPEG compression instead of motion compensation to correct these errors. I use detail blocks for the same reason. Since my detail blocks are not set on a fixed 16×16 grid, I hope to use fewer blocks without affecting the quality of the video sequence.

Detail blocks are allocated to the most noticeable errors first. The error measure I chose gives the luminance of a pixel equal weight with its chrominance. Pixels are sorted by error and covered in turn. The 8×8 block with the greatest error that contains the highest error pixel is selected. I am unconcerned with detail block overlap because each block will write the same colour to the pixels in the overlap section. As a result, even if the blocks are drawn in a different order than I

expect, they will not introduce error. However, I prefer non-overlapping detail blocks, since they maximize the amount of area corrected. I attempt to reduce detail block overlap by setting the pixel error to a negative value once the the pixel has been covered by a detail block.

6.2.3 Significant Differences from MPEG

In this section I will highlight the differences between the MPOG compression scheme and the MPEG compression scheme. In addition I will explain the reasoning behind the different choices and the effects of each choice.

The first difference between the MPEG and MPOG compression schemes is the underlying format of the produced stream. MPEG uses a bit stream so as to increase the compression ratio, while MPOG uses a byte stream to reduce the decompression complexity and remove some of the work load from the CPU.

Discussing the motion compensation portion, there are two significant differences between the two coding techniques. The first difference is that MPEG allows a higher level of precision in selecting a motion block. While MPOG only allows pixel resolution, MPEG allows half-pixel resolution when selecting the source block. This increased resolution can result in motion blocks that more accurately reflect the motion. The second difference in motion compensation is that MPOG allows greater latitude in the size of blocks, and does not operate on a fixed grid. MPEG only uses 16×16 motion blocks and uses a 16×16 grid to locate their destinations. MPOG allows a range of block sizes, allowing both the height and width to range from 6 pixels to 64 pixels. The result of this difference is that MPOG may result in a more compressed representation, since it can potentially use fewer motion blocks.

In both schemes, a correction is applied after the motion blocks have been added to the frame. In MPEG, this correction is an error difference block that is added to the motion block. MPEG uses error differences because they can be compressed further than just pixel values. While pixel values might have any colour, MPEG expects error terms to be close to zero for most pixels. In MPOG, the correction is a set of detail block used to cover the most noticeable errors in the recreated frame. MPOG does not use error terms because adding error terms in graphics hardware

can not be done simply. While it is straightforward to add a positive error term to pixels, adding a negative error term is not directly possible. The accumulation buffer is the only method for adding a negative colour value to a positive colour value, and using it requires too much time. Further, error terms only present a coding savings when combined with a bit stream technique such as Huffman coding, which I have decided to forego. MPOG uses detail blocks to correct errors. Detail blocks are not added to the underlying pixel values, but instead replace them. This operation can be done quickly on a graphics card. As well, there is no time savings for drawing detail blocks on a fixed pixel grid, so detail blocks may be placed anywhere on the screen.

Finally, MPEG and MPOG differ on which frame they use for motion compensation. The frame used in MPEG can be one that was displayed several frames previously while MPOG always uses the last frame drawn. Since it always uses the last displayed frame, MPOG is less robust and unsuitable for network transport. In an MPOG video each frame must be calculated. In the event of an unexpected slowdown, such as a packet loss in network communication, the animation will be noticeably delayed. An MPEG video player can discard some frames to maintain a desired play rate since not all previous frames are necessary to show the next frame.

Chapter 7

Results

In addition to the bouncing ball animation, which I used primarily as an illustration of the capabilities of the decoder, I encoded two other animation sequences. The first is a scene from an animation named Geromino which was developed within the Computer Graphics Lab by a group of students. The scene from Geromino features several dominoes spread across a table. In the animation, the camera pans to the right while smoke rises from a cigarette. I chose this animation to represent a typical, if slightly simple, animation. The second animation I chose was created by Josée Lajoie of the Computer Graphics Lab. It was created to display non-photorealistic rendering techniques. Specifically, objects in this animation are rendered with textures meant to simulate the effect of painting brush strokes. The scene consists of several trees on a hill under a cloudy sky. The animation consists of the camera circling the scene. I chose to use this animation as a difficult test of my encoding algorithm. The non-photorealistic images are less block structured than regular images, and the circular camera motion is not typical and is poorly captured by the linear motion assumption. I tested these three animations against three compression techniques. The first technique used a brute force algorithm for locating motion blocks, the second technique used a hierarchical approach for locating the upper right corner of a motion block, and the third technique was MPEG encoding, which I include to provide a baseline comparison. The results of these tests are summarized in Table 7.1.

Sequence	Play Rate (fps)	Avg. Number Motion Blocks	Avg. Number Detail Blocks	Bits Per Pixel	Avg. Mean Square Error Per Frame
Ball	30.1	8.9	58.1	0.77	6.88
Trees	15.9	91.1	200.0	1.67	762.80
Geromino	15.9	91.3	200.0	1.41	202.05
Hier Ball	30.1	4.75	61.72	0.81	6.94
Hier Trees	15.9	59.6	200.0	1.66	623.45
Hier Ger.	15.8	58.1	200.0	1.41	95.71
MPEG Ball	10.9	-	-	0.29	56.99
MPEG Trees	7.7	-	-	0.19	544.03
MPEG Ger.	7.9	-	-	0.17	101.1

Table 7.1: Statistics for the sequences used. The top section lists results from the brute force encoder, the middle section lists results for the hierarchical encoder, and the bottom section shows the results for the same sequences under MPEG encoding.

The first animation I tested the two encoders against was the bouncing ball animation. I have not included sample images output from the stream created by the automatic encoder since the video player output and the original frames are indistinguishable. Sample images from this animation are displayed earlier in this thesis in Figure 5.1. The animation plays at a steady rate of approximately 30 frames per second. The brute force encoder and the hierarchical encoder are equivalent and both compare well with the simplified encoder described in Chapter 5. The slight increase in Mean Square Error (MSE) over what the simplified encoder achieved can be attributed to a greater number of update frames between each JPEG encoded base frame in the simplified encoder's video stream. Also note that both examples compare well with MPEG encoding; although neither encoding is compressed as much as the MPEG encoding, they both play at nearly 3 times the rate of the MPEG video.

The second animation I tested the two encoders against was the non-photorealistic animation. The animation sequence consisted of 33 frames, each 480×480 pixels. In Figure 7.1 you can see the original frames on the left, frames from the brute force encoder in the middle and frames from my hierarchical decoder on the right. The frames choosen were taken six, ten and fourteen frames after a JPEG frame in the video stream. Since I use a high quality setting on my JPEG encoded frames they have few noticable artifacts, and are not noteworthy. The decoder played the video at a constant rate, but the playback speed was slower than desired. The playback

took 2.06 seconds, which translates to 15.9 frames per second. The animation required 1.66 bits per pixel. The hierarchical encoder did not reduce the time to encode frames significantly for this animation, with each frame taking approximately 80 minutes. However, it appears to have found higher quality motion blocks, using only 59 motion blocks on average while the brute force algorithm used 91 motion blocks on average. As a direct result the amount of error in the hierarchical encoder is reduced by 18%. The animation, as encoded by both encoders, suffers from two noticeable artifacts. The first artifact is a trail of pixels left behind by moving objects. The second artifact was blockiness where motion blocks were applied. An MPEG encoding of the same 49 frames required only 0.18 bits per pixel and had fewer artifacts in the frames. However, the playback rate, when forced to play every frame, was approximately 8 frames per second, or half the playback rate of the MPOG encoder.

The third animation I tested the two encoders against was the sequence from the Geromino animation. The animation consisted of 33 frames, each 512×480pixels. In Figure 7.2 you can see the original frames on the left, frames from the brute force encoder in the middle and frames from the hierarchical decoder on the right. The frames choosen were taken six, ten and fourteen frames after a JPEG frame in the encoded stream. Also note that the frames were gamma corrected after being captured from the video stream to lighten the images and improve visibility in the printed copy. As in the previous example, both the brute force algorithm and the hierarchical algorithm produce streams that play at the same rate and have the same compression ratio. Also like the previous example, the hierarchical algorithm picks fewer motion blocks and has a lower error. Unlike the previous example, the hierarchical encoder greatly improved the run time. While the brute force algorithm required over 30 minutes to encode an update frame, the hierarchical encoder averaged 12 minutes per update frame. The artifact that is most noticable in this example results from motion blocks not being constrained to hold an entire moving object. As a result, the dominoes can be seen to decay, especially once the frame has been gamma corrected. Compared to MPEG encoding the same advantages and disadvantages are evident. While MPEG achieves a better compression ratio, my two encoders have a better playback rate.

Since I am compressing a sequence of captured frames, I am not concerned with the running

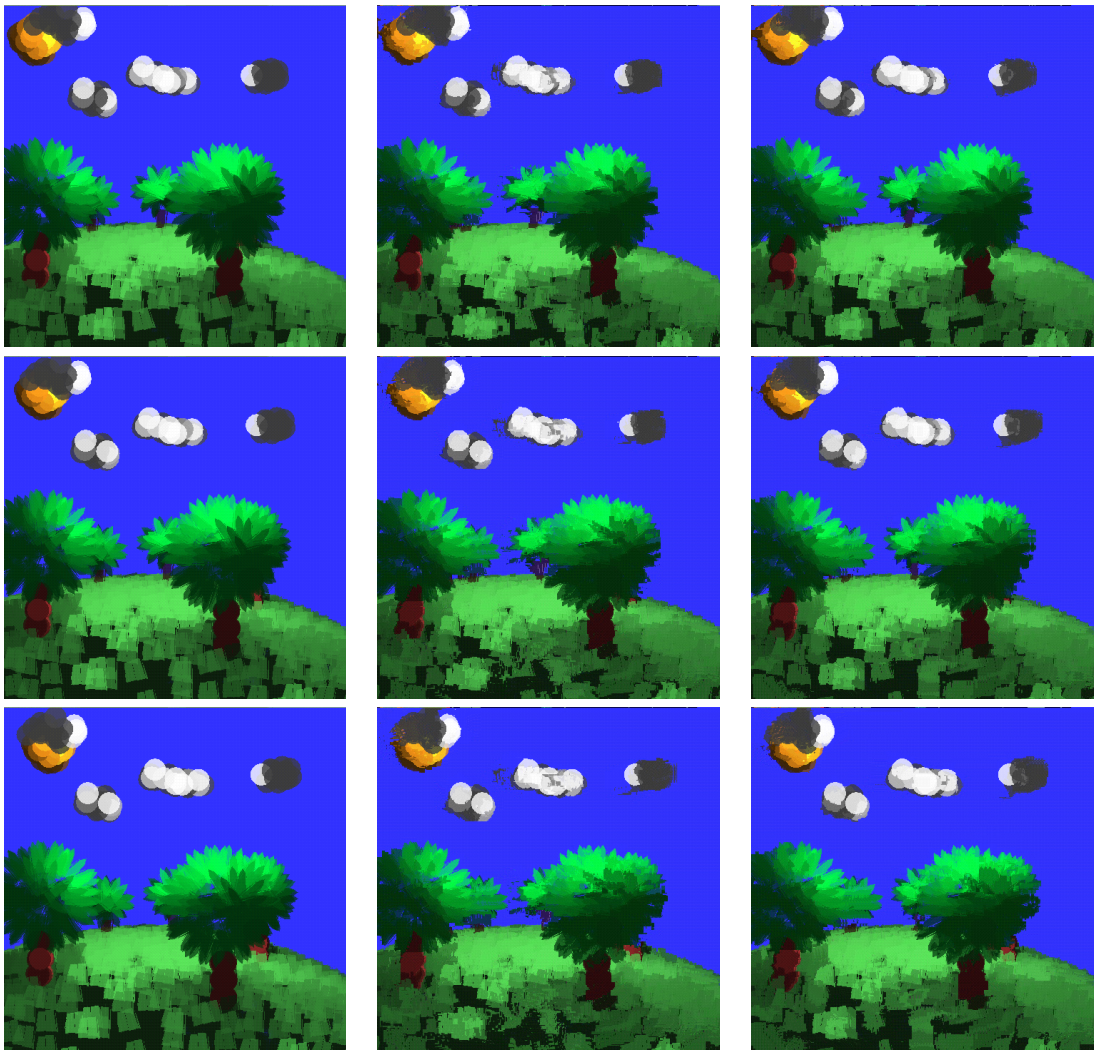

Figure 7.1: Sample images taken from the non-photorealistic animation, with original frames on the left, frames from the brute force encoder in the centre and frames from the hierarchical encoder on the right.

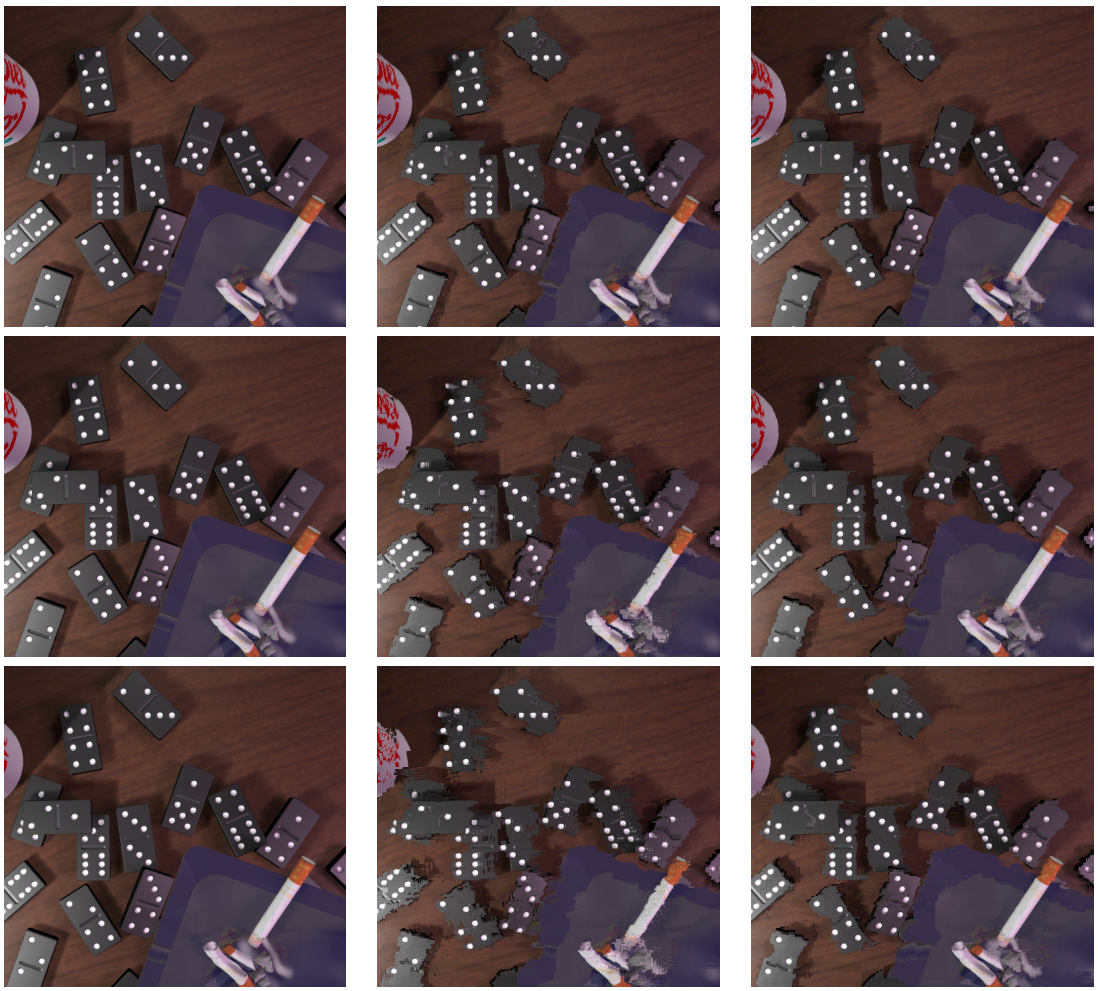

Figure 7.2: Sample images taken from the Geromino animation, with original frames on the left, frames from the brute force encoder in the centre and frames from the hierarchical encoder on the right.

time of the encoder. However, to make the encoder applicable to live video, optimization of the motion block detection phase will be required. Currently, encoding frames from the Geromino animation require approximately twelve minutes each, with over 99% of all time spent detecting motion blocks.

Chapter 8

Conclusions

I proposed an alternate approach for streaming video. Rather than using uncompressed video which requires bandwidth resources that few computers possess, or decompressing video with the CPU or a custom card, I proposed that a method could be devised to stream video principally using the graphics card. This streaming method would have the advantage of not requiring the computer to send the decompressed video stream across any system buses since the decompression would occur directly on the graphics card. My initial compression schemes do not fully achieve this goal since base images are decoded by the CPU and detail blocks are not compressed. However, there is some expectation that more recent graphics hardware would permit operations that would eliminate these limitations.

The MPOG compressed stream suffers from two noticable artifacts, especially when the entire frame is in motion. The first artifact is trails left behind an object in motion. These are likely caused by the problem of occlusion; the material behind the object in motion can not be represented by a motion block and there are two few detail blocks to cover all errors. The second artifact is blockiness where motion blocks are placed. Inside the motion block we know that most pixels are suitable replacements. However, outside a motion block the pixels may still be incorrect. Thus the corrections made will tend to highlight the pixels that could not be fixed. One possibility would be to use more detail blocks to cover the incorrect pixels. Another solution is

to blend incoming pixels with existing colour values in a frame at the edge of the motion block. While this may result in greater amount of error, it will reduce the coherent error, making the underlying block structure less noticable.

8.1 Future Work

One of the significant setbacks for this thesis was the failure to develop a method for performing image decompression techniques on the graphics card. By using OpenGL 1.2, I expect to be able to create such a technique. OpenGL 1.2 introduces the notion of multitexturing, which allows two or more textures to be blended together. This could be used to perform matrix multiplications, and might make singular value decomposition decompression techniques, such as the one described in [YL95], possible on the graphic card. This technique would allow compression of detail blocks and would greatly improve compression rates.

To increase the portability of this encoding scheme, it should be ported to run on a personal computer. As CPU speeds and graphics cards continue to improve, PCs are becoming a viable alternative to workstations for many tasks. There are only two requirements which restrict my encoder and decoder to sgi computers. First, the encoder uses sgi's Image Format Library (IFL). I use IFL to read in image files and translate them into a raw array of pixel values. IFL can read images from most, if not all, commonly used image formats. I could replace IFL either by translating all images into one format, either a simple one to read, such as PPM, or by translating into one format that can read using a system library, such as PNG. The second requirement is an OpenGL extension that the decoder uses to determine when a monitor screen refresh has occurred. Since this extension seems not to work well under high load, replacing it with the system clock might be beneficial.

Bibliography

[Auz94] Valliere Richard Auzenne. *The Visualization Quest.* Fairleigh Dickison University Press, Cranbury, NJ, 1994.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[COMF99] Daniel Cohen-Or, Yair Mann, and Shachar Fleishmann. Deep compression for streaming texture intensive animations. *Computer Graphics (SIGGRAPH '99 Proceedings)*, pages 261–267, 1999.

[Eve00] Cass Watson Everitt. High-quality, hardware-accelerated per-pixel illumination for consumer class OpenGL hardware. Master's thesis, Mississippi State University, May 2000.

[FGW97] Borko Furht, Joshua Greenberg, and Raymond Westwater. *Motion Estimation Algorithms for Video Compression.* Kluwer Academic Publishers, 1997.

[Lev95] Marc Levoy. Polygon-assisted JPEG and MPEG compression of synthetic images. *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 21–28, 1995.

[Sch93] M. Schwenden. *Onyx^{TM}Deskside Owner's Guide*, 1993.

[Tek95] A. Murat Tekalp. *Digital Video Processing.* Prentice Hall, Upper Saddle River, NJ, 1995.

[Wal91] Gregory K. Wallace. The JPEG picture compression standard. *Communications of the ACM*, April 1991.

[Wat95] John Watkinson. *Compression in Video and Audio.* Focal Press, 1995.

[WND97] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide.* Addison-Wesley Developers Press, 1997.

[YL95] Jar-Ferr Yang and Chiou-Liang Lu. Combined techniques of singular value decomposition and vector quantization for image cosing. *IEEE Transactions on Image Processing*, pages 1141–1145, 1995.