(46)

Adaptive Degradation of Images in Network Applications

by

Simon Lau

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 1998

Copyright 1998 Simon Lau. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ______________________________
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by ______________________________
V. Michael Bove, Jr.
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 14 1998

LIBRARIES

Eng.

Adaptive Degradation of Images in Network Applications
by
Simon Lau

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1998

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The all-or-nothing image download scheme currently used on the World Wide Web is unacceptable. This scheme is entirely too rigid, considering that for a given image, the same amount of data is sent to every client who requests the image, with no consideration of how fast this data can be downloaded or the available processing power at the client. This thesis proposes an adaptive image degradation model in which images are scaled "on-the-fly" so that the amount of data sent is in relation to the network and processing capability of the requesting client. The primary contribution of this thesis is the development of a wavelet-based adaptive image viewing system (WAIVS) using Netscape plug-in technology, CGI scripts, and a highly scalable wavelet-based image coder/decoder developed at Texas Instruments,. The benefits of such an adaptive image viewing system are more efficient use of bandwidth and reduced download time as compared to the currently used system.

Thesis Supervisor: V. Michael Bove, Jr.
Title: Associate Professor of Media Technology, The MIT Media Laboratory

ACKNOWLEDGMENTS

I would like to thank Raj Talluri and Raja Rajasekaran for giving me the opportunity of joining the Video Technology Branch of the Media Technology Lab at Texas Instruments, Dallas, Texas, for my thesis work. I am especially grateful to Prof. V. Michael Bove, whose invaluable advice and support were essential to the successful completion of this thesis. I also would like to thank Jie Liang, Jonathan Courtney, and Jennifer Webb for offering me ideas and sound technical advice. Special thanks to Jon Orwant for answering my Perl questions despite his very busy schedule. My heartfelt thanks to Fred Hill, Jr., for proofreading my writing and getting me through the days when finishing the thesis seemed impossible. To my dearest friends Eric Natale, Kamal Swamidoss, Lajos Molnar, Xiaozhen Zhang, Rushani Wirasinghe and Florence Zhu, thank you for your encouragement and support. To my father and grandmother, thank you for your special phone call from Hong Kong and your words of wisdom. I miss you both very much. And last but not least, to my mother and my best friend, Blaise Lin, this thesis is dedicated to you. Thank you for putting up with yet another semester of my busy schedule. I love you very much.

Table of Contents

1. INTRODUCTION **6**

2. ADAPTIVE IMAGE DEGRADATION MODEL **9**

2.1 SYSTEM MODELS 9

2.2 DESIGN CRITERIA 11

3. WAVELET IMAGE CODING **14**

3.1 OVERVIEW OF WAVELETS 14

3.1.1 Mathematical Description *15*

3.1.2 Wavelet Decomposition of Images *16*

3.2 SHAPIRO'S EMBEDDED ZEROTREE WAVELET ALGORITHM 17

3.3 THE PREDICTIVE EMBEDDED ZEROTREE WAVELET (PEZW) IMAGE CODEC 20

4. WAVELET-BASED ADAPTIVE IMAGE VIEWING SYSTEM (WAIVS) **23**

4.1 SYSTEM OVERVIEW 23

4.2 TI PEZW CODEC SOFTWARE 25

4.3 CLIENT-SIDE DATA PROCESSING: THE NPWVT PLUG-IN 26

4.3.1 The Plug-in Life Cycle *28*

4.3.2 The Plug-in Object Model *29*

4.3.3 The NpWvt Plug-in Instance Structure *30*

4.3.4 The NpWvt Plug-in Methods *30*

4.3.5 The Subclass Function *34*

4.3.6 MIME Types and The <embed> Tag *35*

4.4 SERVER-SIDE DATA ADAPTATION 37

4.4.1 CGI Basics 37

4.4.2 Perl Script 38

4.4.3 The extract Program and the WVT Bitstream Syntax 40

4.5 SAMPLE EXECUTION OF THE SYSTEM 44

5. SYSTEM EVALUATION 47

5.1 TIME EFFICIENCY 47

5.2 SERVER LOAD ISSUES 49

5.3 BENEFITS OF ADAPTIVE DEGRADATION 50

5.3.1 Better Viewing Experience for the Users 50

5.3.2 Ease of Use for Content Creators 50

5.3.3 Elimination of Wasted Bandwidth 51

6. CONCLUSION 53

7. FURTHER WORK 54

7.1 PER OBJECT ADAPTIVE DEGRADATION 54

7.2 PROGRESSIVE SCANNING 55

7.3 EXTENSION TO VIDEO 57

8. BIBLIOGRAPHY 58

9. APPENDIX 60

9.1 COMPARISON OF ENCODING QUALITY BETWEEN WAVELETS AND JPEG 60

9.2 NPWVT.CPP 64

9.3 NPH-ADAPT.PL.. 71

9.4 EXTRACT.C.. 71

9.5 AKIYO.HTML 79

1. Introduction

The World-Wide-Web (WWW) is an increasingly popular medium for information exchange and communication. One major problem with Web surfing today remains that many Web pages take a long time to load, due to the limitation in bandwidth and the heavy use of images and other types of media (e.g., graphics, audio, video). The conventional model of viewing images over the Web is very rigid in that the same image file, hence a fixed volume of data, is transferred to any requesting client regardless of the available bandwidth. Thus, users with low-bandwidth connections generally have to wait longer for images to download. In the extreme case, frustrated users may choose to disable the image loading option of the browser altogether so that the remaining text portion of a Web page would load quickly.

An alternative to this "all-or-nothing" download paradigm is to adaptively scale an image to a different level of *information resolution* before it is sent to the client based on available bandwidth. Information resolution refers to the content-richness of an image. It can be described more concretely by other measures, such as picture quality and image size. For example, a smaller version of an image is considered to be at a lower information resolution as compared to the original image. A lower information-resolution version of an image generally requires fewer bits to represent and, therefore, imposes a lower bandwidth requirement. In this adaptive model, lower information-resolution versions of images are sent to clients who have low-bandwidth connections, thus shifting the penalty in download speed associated with the "all-or-nothing" download paradigm to a tradeoff in information resolution.

This thesis proposes a model for adaptive media content delivery over the Web, focusing primarily on the adaptive degradation of images. The term "degradation" is used because the model is based on storing the original images (at full information resolution) at the server, and deriving the lower information resolution versions from these originals. An image is said to be adaptively degraded, if necessary, before the server sends it to the client who requested the image. The contribution of this thesis is the development of a wavelet-based adaptive image viewing system (WAIVS) based on this adaptive image degradation model. WAIVS is constructed using Netscape plug-in technology, CGI scripts, and a highly scalable wavelet-based image codec developed at Texas Instruments. The benefits of WAIVS are efficient use of bandwidth and timely delivery of image data to clients with a wide range of bandwidth capabilities.

Note that the general concept of adaptive degradation extends beyond images to other types of media, including audio and video. In [1], Bove describes a scalable open architecture television system in which the resolution and frame rate of the display device need not match that of the originating source. A digital image representation that is scalable in resolution and frame rate is used in this system so that an NTSC (525 lines, 60 fields per second) receiver can display a signal originating with a higher definition and frame rate, and vice versa. An additional goal of this work is, therefore, the establishment of a general adaptive degradation framework, which is extensible to other types of media.

Chapter 2 presents the adaptive image degradation model. Chapter 3 gives an overview of the wavelet representation of images, the embedded zerotree wavelet (EZW) algorithm, and the TI predictive embedded zerotree wavelet (PEZW) image codec. Chapter 4 describes the implementation of WAIVS. Chapter 5 evaluates WAIVS and

suggests improvements to the system. Chapter 6 gives some concluding remarks. In closing, Chapter 7 proposes further research problems based on the result of this work.

2. Adaptive Image Degradation Model

2.1 System Models

The most common way of viewing images over the Web follows the "all-or-nothing" image download model, as illustrated in Figure 1. At the server, a single representation (JPEG and GIF being the most popular image file formats) is stored for each image. Upon request, the whole image file is transmitted across the network. This model is very rigid because the amount of transferred data for a particular image is fixed.

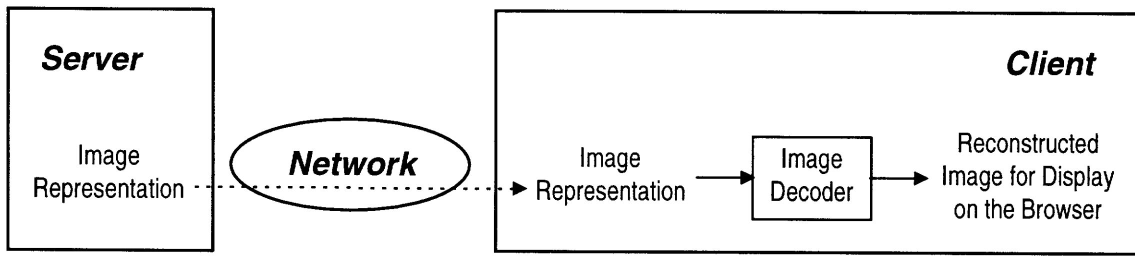


Figure 1: "All-Or-Nothing" Image Download Model

Instead of having only one representation of each image, some Web sites support thumbnails, or smaller-size versions of the original pictures. A user can then preview an image before he decides to download the full-size original. This scenario exemplifies the adaptive image selection model, which is shown in Figure 2. More generally, several versions of the same image are stored at the server. Upon request, one of these versions is selected to be transmitted back to the client. The selection is based on some hints originating from the client, which contain information about the user's preference, the client's processing power and display capability, connection bandwidth, etc. This model is less restrictive than the existing one because the amount of transferred data for a given

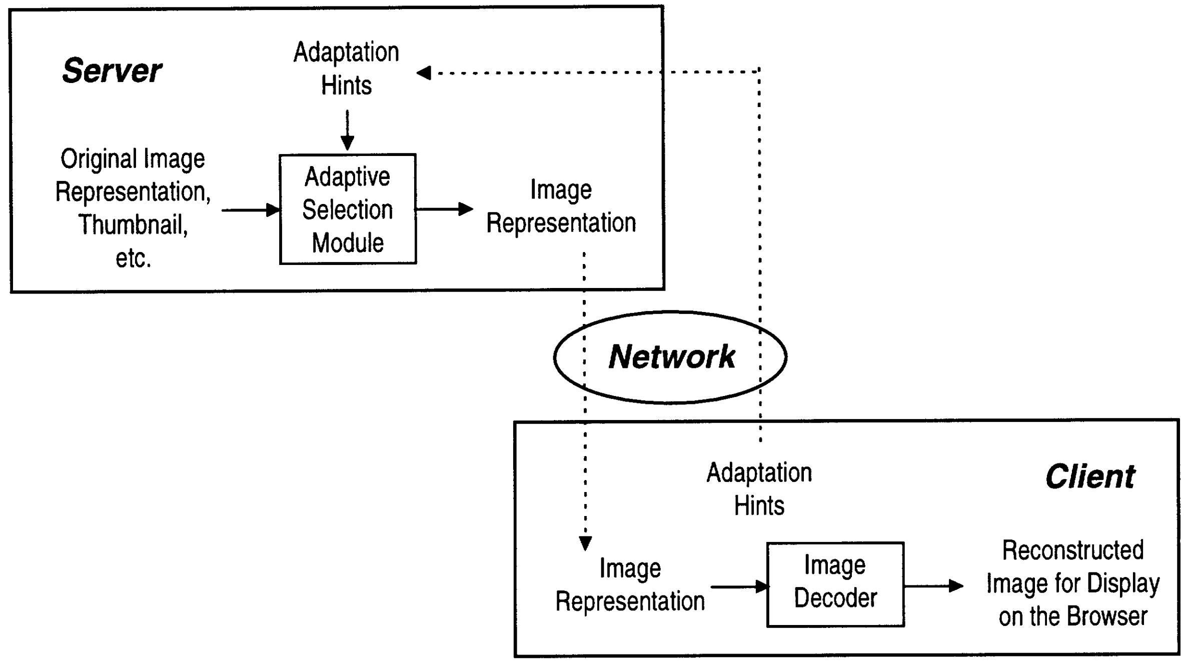


Figure 2: Adaptive Image Selection Model

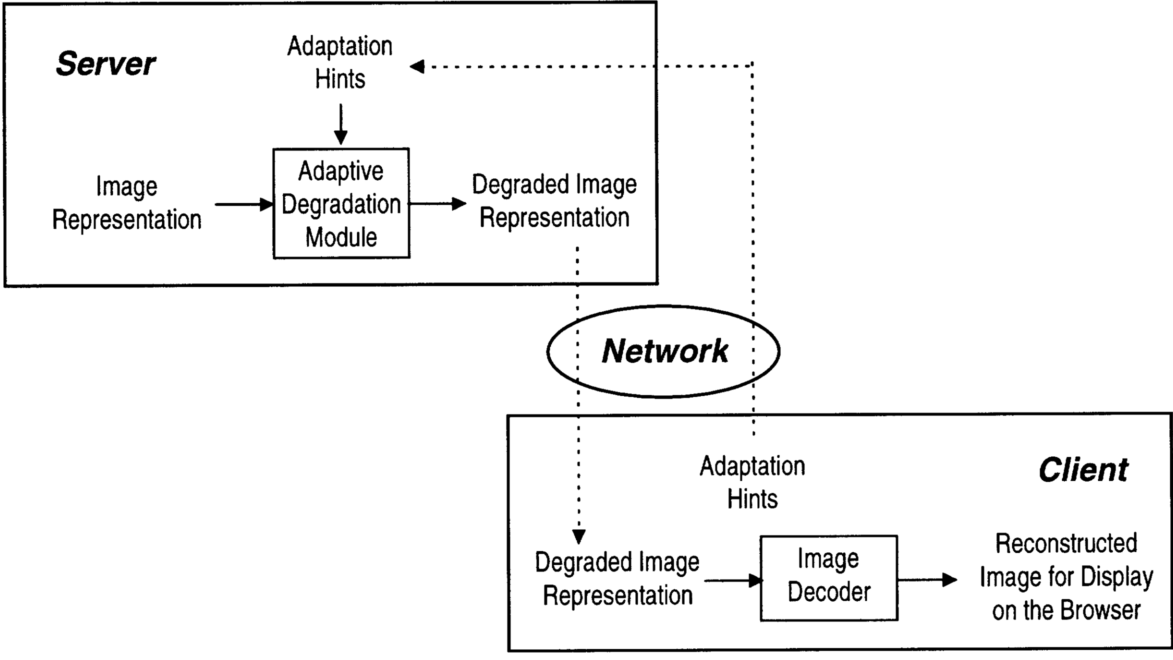


Figure 3: Adaptive Image Degradation Model

image is no longer fixed. However, the flexibility of adjusting the data rate comes with a cost of keeping multiple versions of each image at the server. The additional storage requirement can be very large if many different versions are desired. Also, generating all the different versions can be quite cumbersome.

The adaptive image degradation model, depicted in Figure 3, takes the best property of each of the two previously mentioned models and combines them. Similar to the adaptive image selection model, when a client requests a particular image, a different version of the image is transmitted depending on the adaptation hints provided by the client. Unlike the selection model, the different versions are generated on-the-fly by an adaptive degradation module. Thus the server only needs to keep one copy of the image (i.e. the original image representation), as in the "all-or-nothing" image download model. However, the added complexity in the adaptive degradation process comes at a cost of achieving an adaptive data rate without extra storage at the server. As we will discuss later in greater detail, when wavelet-based image coding technology is applied, image degradation becomes a straight-forward process. The added complexity is, therefore, minimized and the benefits of an adaptive image degradation system outweigh the costs.

2.2 Design Criteria

When designing a system based on the adaptive image degradation model, the following issues should be considered: the choice of image characteristics on which adaptation is performed, the source of adaptation hints, and the complexity of the adaptive degradation process.

The adaptive image characteristics should be chosen such that degradation in these characteristics results in not only a data rate that is reduced but also a picture that looks acceptable. Picture quality and image size, therefore, are two reasonable adaptive characteristics. For most image formats, the smaller the image, the smaller the file size. For the JPEG format, a lossy DCT-based compression format, an image encoded at a lower quality factor also requires fewer bits to represent. In general, a smaller or slightly blurry picture is often acceptable. If the user only requires a rough idea of the image content, reducing the quality or size of the picture is certainly acceptable. If the client display resolution is very low so that the downloaded images appear too big to fit inside the browser window, shrinking the images before they are transmitted to the client may even be desirable. This could be useful in portable, palm-held computing devices.

The adaptive degradation module bases its operation on adaptation hints provided by the client. These hints can be either specified by the user or automatically generated. The choice depends on which of user-configurability or user-abstraction is preferred. For user-specified hints, the user can be given a range of degradation levels for each adaptive image characteristic supported by the system. The user may, for example, select the degradation levels in a "Preferences" menu of the browser. To automatically generate adaptation hints, additional monitoring modules are necessary to observe certain client conditions, such as connection bandwidth, browser window size, etc. The monitored values are then translated into an appropriate level of degradation for each adaptive image characteristic to be sent to the server.

In order to generate a degraded version of an image on-the-fly, the adaptive degradation process cannot be too complex or take too long to complete. This constraint,

in turn, limits the choice of image format. The file syntax for the image format must be parsable on-the-fly so that the relevant data associated with the degraded version of an image can be easily extracted from the original image representation. The JPEG format is one counterexample. Suppose image quality is chosen to be the adaptive characteristic and JPEG is the image format. To degrade the image quality of a JPEG file on-the-fly, the adaptive degradation module cannot simply select a subset of bytes from the original JPEG file and compact the bytes into a smaller file to be sent to the client. A file generated in this manner is not a decodable JPEG file. Instead, the adaptive degradation module must first decompress (decode) the original JPEG file stored at the server into raw image data, then compress (or encode) the raw data back to JPEG format using a lower quality factor. This degradation process is not time efficient at all. Moreover, since JPEG is a lossy compression, the decompression/compression sequence in the degradation process causes even more data loss, resulting in additional degradation in the transmitted picture. On the other hand, an image encoded with a wavelet-based image codec has a file format that makes on-the-fly adaptive degradation of image quality and picture size possible. The TI Predictive Embedded Zerotree Wavelet (PEZW) image codec produces such a file format and will be described in the next chapter.

3. Wavelet Image Coding

A wavelet-based image codec is used to encode images to be stored at the server. Section 3.1 gives an overview of wavelet theory and the wavelet representation of images. Section 3.2 talks about Shapiro's embedded zerotree wavelet (EZW) algorithm for a more compact representation of the image subbands. Section 3.3 introduces the predictive embedded zerotree wavelet (PEZW) image codec developed at Texas Instruments.

3.1 Overview of Wavelets

Wavelets are a fairly recent and quite well accepted method of compressing images. They have been shown to provide encoding quality superior to transform coding techniques based on the DCT (discrete cosine transform), as used in the JPEG compression standard, on still images at equal bitrates (see [4] and the Appendix, Section 9.1). Since the wavelet transform operates over the entire image whereas DCTs typically operate on 8x8 blocks, the blocking artifacts commonly produced by DCT schemes at low bitrates are absent in wavelet based image compression. The superior quality at low bitrates also stems from the fact that basis functions are chosen to cover a wide range of signal behaviors. Thus, wavelets provide a signal representation in which some of the coefficients represent signal behavior that is more localized in the space domain and tends to be wide band in the frequency domain (e.g., edges) and some of the coefficients represent signal behavior that is more localized in frequency but persist over a wide area in space (e.g., smooth regions). Efficient algorithms exist for coding wavelet coefficients so that very high compression ratios can be achieved.

3.1.1 Mathematical Description

Wavelets are families of functions generated by dilations and translations from a single function, $\Psi(t)$, called the mother wavelet. The dilations and translations can be chosen as powers of 2 to produce the following wavelets:

$$\Psi_{m,n}(t) = 2^{-m/2}\Psi(2^{-m}t - n)$$

These wavelets can be used to describe arbitrary functions, such as image data. Functions are expressed in terms of a superposition of the basis functions $\Psi_{m,n}$:

$$f(t) = \sum_{\forall m, \forall n} c_{m,n}(f)\Psi_{m,n}(t)$$

The mother wavelet Ψ is often designed so that the resulting wavelets $\Psi_{m,n}$ form an orthogonal basis. Thus the computation of $c_{m,n}(f)$ simplifies to the following equation:

$$c_{m,n}(f) = \int \Psi_{m,n}(x) f(x) dx$$

This equation is known as the continuous-time wavelet transform. The set of transform coefficients $c_{m,n}(f)$ is a complete representation of f. They are the weights for the individual basis functions which compose the complete signal.

Each mother wavelet has an associated scaling function $\Phi(t)$ which can also be written in terms of dilations and translations:

$$\Phi_{m,n}(t) = 2^{-m/2}\Phi(2^{-m}t - n)$$

At a given resolution n, the scaling function Φ supplies low-frequency information about the signal, whereas the wavelet function Ψ furnishes more detail about the signal in a complementary fashion.

3.1.2 Wavelet Decomposition of Images

When wavelet techniques are extended into the discrete domain, for image compression applications for example, the wavelet and scaling functions are described in terms of filter taps, rather than continuous functions. In image compression, wavelets filters, such as the popular Daubechies filters, are used to decompose images into various frequency bands. The decomposition occurs in both the horizontal and vertical directions using one-dimensional wavelet filters H and L which corresponds to Ψ and Φ, respectively. Figure 4 shows the diagram for one level of wavelet decomposition. The end result is four subbands: LL (low-frequency subband), LH, HL, and HH (three high-frequency subbands).

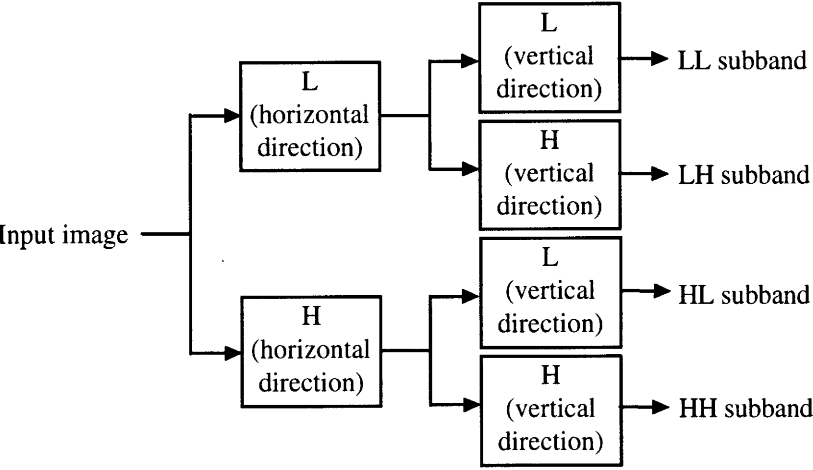


Figure 4: One-level wavelet decomposition of an image into four subbands

Wavelet compression schemes generally use three to five levels of wavelet decompressions. Typically the LL subband is recursively decomposed. The decomposition process satisfy the critical sampling property, preserving the total number of samples. This is illustrated in Figure 5.

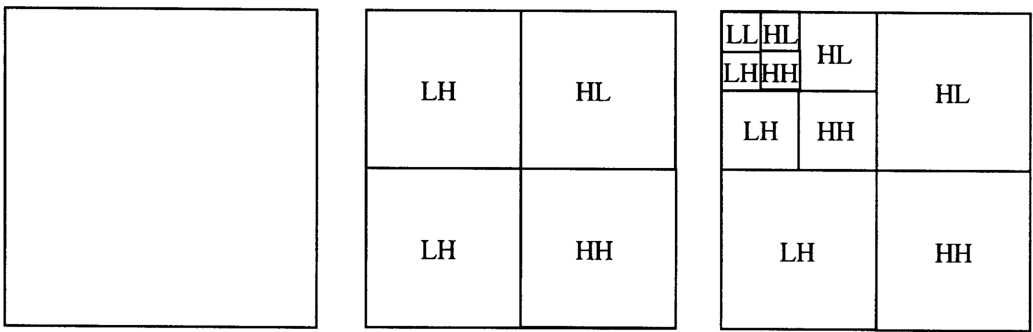


Figure 5: Wavelet decompositions of images: original image (left), 1-level decomposition (middle), 3-level decomposition (right).

For more details on the wavelet mathematics, the reader should refer to [11]. Easy-to-read tutorial articles on wavelets and their use in image compression can be found in [4], [7] and [12].

3.2 Shapiro's Embedded Zerotree Wavelet Algorithm

Performing the wavelet decomposition itself does not compress the images. The compression comes from efficiently coding only those wavelet coefficients which contributes the most energy and discarding the insignificant ones. Note that we need to encode not only the value of the significant coefficients but also their positions. In the literature ([8], [9]), the position information of the significant coefficients is referred to as the *significance map*. Thus the true cost of encoding the significant coefficients is:

Total Cost = Cost of Significant Values + Cost of Significance Map

Shapiro's embedded zerotree wavelet algorithm (EZW) greatly reduces the cost of coding the significance map by exploiting the relationship between the LL subband, which contains most of the significant information, and the other subbands, which contain primarily edge information. Shapiro's celebrated paper on his EZW algorithm can be found in [9]. High-level descriptions of the EZW algorithm can be found in [5] and [8].

After a wavelet subband decomposition, it is readily observed that if the magnitude of a wavelet coefficient in the LL subband is below a certain threshold T, then it is very probable that wavelet coefficients in the corresponding positions of the higher frequency subbands will also have magnitudes less than T. Suppose we call a coefficient in a lower resolution subband a "parent" and the corresponding coefficients in the next higher resolution subband "children." Then parent-children relationships are illustrated in Figure 6.

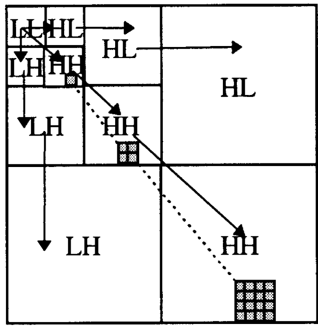


Figure 6: Parent-children dependencies of subbands

Shapiro's algorithm basically takes advantage of this *self-similarity* property of coefficients across subbands. It begins by sending a significance map of the data indicating the locations of the important coefficients, according to a given threshold, and supplying a rough approximation to these coefficients. This step is called the dominant pass. Next, addition information is sent to refine the approximations of the important coefficients. This step is called the subordinate pass. Then the threshold for determining the importance of a coefficient is lowered and the process is repeated. Another dominant pass is sent with information about less important coefficients and the coefficient representations are further refined with another subordinate pass. This process repeats

until encoding is stopped, typically when a target bitrate is reached. All resulting data is further compressed using arithmetic coding.

In creating the significance map, each coefficient is classified as one of the following categories:

1. Zerotree root (ZTR): the coefficient and all descendants are insignificant with respect to the threshold T.
2. Isolated zero (IZ): the coefficient is insignificant but has at least one significant decedent.
3. Positive significant (POS): the coefficient is significant and greater than zero.
4. Negative significant (NEG): the coefficient is significant and less than zero.
5. Predictably insignificant: descendant of a zerotree root.

During a dominant pass, only the symbol (ZTR, IZ, POS, or NEG) corresponding to the category of each coefficient is sent. The coefficients are scanned in a particular order, where all parents are scanned before the children, as shown in Figure 7. Note that when a coefficient is predictably insignificant, no symbol is sent because the locations of coefficients of this category can be derived from zerotree roots. This is where many bits are saved because only one symbol, the zerotree root, is necessary to represent an entire tree of insignificant coefficients (called the zerotree).

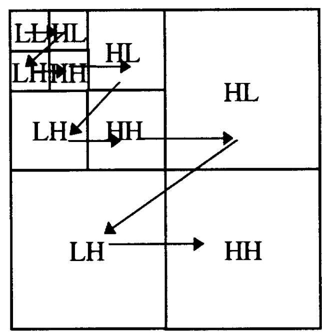


Figure 7: Scanning order for encoding significance map

3.3 The Predictive Embedded Zerotree Wavelet (PEZW) Image Codec

Shapiro's work has been shown to be one of the most efficient image coding methods around. Many researchers have since worked on variations of the original zero-tree method. Liang, at Texas Instruments, proposed and implemented a novel wavelet image coding algorithm that improves the results of the EWZ-based algorithms through improved symbol set for zerotree encoding, greater base band compression with DPCM (Differential Pulse Code Modulation) coding, and better context modeling for arithmetic coding. This section only describes the improved symbol set. For discussion on the other two improvements, see [6].

In creating the significance map, each coefficient is classified as one of the following categories:

1. Zerotree root with insignificant value (ZTRZ): the coefficient and all descendants are insignificant with respect to the threshold T, same as ZTR in Shapiro's algorithm.
2. Zerotree root with significant value (ZTRS): the coefficient is significant but all descendants are insignificant with respect to the threshold T.

3. Isolated zero (IZ): the coefficient is insignificant but has at least one significant descendant.
4. Isolated significant (IS): the coefficient is significant but not all of its descendants are insignificant.
5. Predictably insignificant: descendant of a zerotree root.

In comparison to the original symbols, POS and NEG is replaced with one symbol IS because the possibility for positive numbers and negatively numbers are about equal. This leaves room for a new symbol, ZTRS, to make significant coefficients permissible as the root of a zerotree. The motivation behind this modification is the hypothesis that a significant coefficient quite often has all insignificant descendants. Theoretical analysis and empirical results support this hypothesis. To encode this scenario using the new symbol set, only a single ZTRS symbol is needed; whereas using Shapiro's symbol set, one IS symbol followed by several ZTR symbols are required.

The Predictive Embedded Zerotree Wavelet (PEZW) image codec developed at Texas Instruments uses this new symbol set. The new method has been shown to have an improvement of 1-3 db in PSNR (peak signal-to-noise ratio) over Shapiro's EZW algorithm for the test images considered [6]. The PEZW encoder produces an embedded bitstream, which means that all encodings of the same image at lower bitrates are embedded in the beginning of the bitstream for the target bitrate. Thus different versions (of varying sizes and qualities) of a particular image can be generated on-the-fly by decoding different portions of the same bitstream. Furthermore, only one version (corresponding to the highest quality) of an image needs to be stored at the server, since all lower-quality versions of an image can be derived from the same bitstream.

4. Wavelet-based Adaptive Image Viewing System (WAIVS)

A wavelet-based adaptive image viewing system (WAIVS) is constructed based on the adaptive image degradation model. Section 4.1 presents an overview of WAIVS and introduces the various components of the system. Section 4.2 gives a brief description of the TI PEZW decoder. Section 4.3 describes the construction of the client portion of the system. Section 4.4 talks about how server-side adaptation is implemented. Section 4.5 illustrates the entire system at work using a sample scenario.

4.1 System Overview

WAIVS builds on the Web's client-server model to create an image viewing environment in which the processing of image data is distributed between a program that executes on a Web server and a plug-in that runs in the browser. The low-level details involved in most client-server systems are handled entirely by the Web server and the browser. Figure 8 shows a high-level overview of the system.

WAIVS consists of the following components:

- `nph-adapt.pl`: a CGI-program written in Perl which interfaces the Navigator browser and the adaptive degradation module, a program called `extract`. It reads in two adaptation hints, `bitplaneOff` and `resolutionOff`, from the `QUERY_STRING` environment variable. Next it calls `extract` with these variables and the name of the wavelet-encoded (WVT) image file. When `extract` returns, the required HTTP response headers are generated and sent to the Web client along with the degraded wavelet-encoded data returned from the `extract` program.

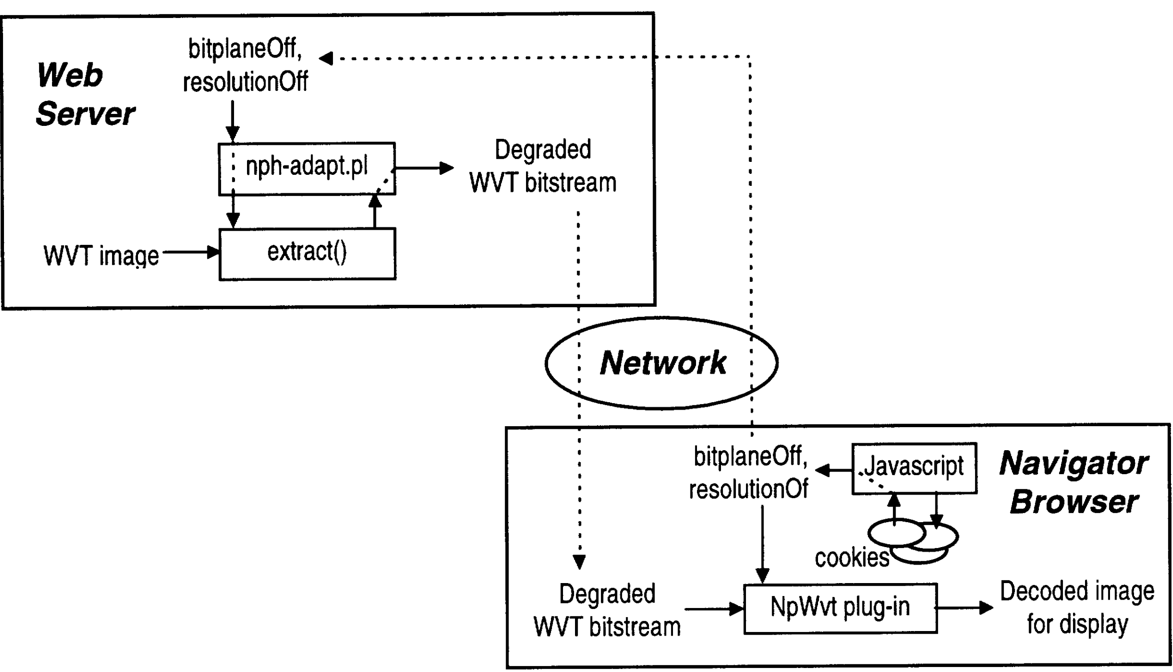


Figure 8: WAIVS System Overview

- `extract`: an executable C program running at the server which adaptively degrades a wavelet-coded (WVT) image. Depending on the two parameters `bitplaneOff` and `resolutionOff`, the program parses the specified WVT file, selects and packs a subset of bytes into a condensed bitstream. The condensed WVT bitstream is returned to the calling Perl program, `nph-adapt.pl`.
- NpWvt plug-in: a Netscape plug-in for decoding the WVT image format and displaying the decoded image on the Web browser. The plug-in resides at the client side and is dynamically loaded by Navigator when data of the MIME type "`application/x-waveletimage`" is downloaded.
- Javascript: it is used to retrieve the adaptation hints `bitplaneOff` and `resolutionOff` from a client side cookie. It passes these hints to the

server by setting up the `QUERY_STRING` field appended to the URL of the Perl program. It also sets up the appropriate name/data pairs in the `<embed>` tag so that the NpWvt plug-in gets these two parameters as well.

The implementation details of each of these components are discussed in the remaining sections of this chapter.

4.2 TI PEZW Codec Software

Jie Liang, at the Media Technology Lab of Texas Instruments, developed the command-line version of the TI PEZW codec software. The input syntax for the PEZW decoder is as follows:

```
decoder inputFile width height [bitplaneOff] [resolutionOff]
```
[1]

Listing 1: Input Syntax for the PEZW Decoder

The required input arguments are the name of the file containing the encoded bitstream, and the dimensions of the original image in pixels. The optional arguments `bitplaneOff` and `resolutionOff` specify the number of SNR scalability layers and spatial scalability layers (see Section 4.4.3), respectively, to skip during the decoding process. A higher `bitplaneOff` setting gives a blurrier picture; whereas a higher `resolutionOff` setting leads to a smaller image size. If these optional parameters are not specified or set to zero, the reconstructed image will be at full quality and resolution. Figure 9 shows some examples of the reconstructed images for various decoder settings.

---

[1] The latest version of the TI PEZW codec software supports many additional options, such as bitrate control, minimum bit budget for the chrominance components, and mosaicing. For clarity of this discussion, only a subset of the available options is included.

The reconstructed images are written out to two files, in YUV(4:2:0) and RGB formats. An external image viewer program which supports these formats (e.g. Paint Shop Pro) is required to view the reconstructed images.

4.3 Client-Side Data Processing: the NpWvt Plug-in

To integrate the TI PEZW decoder into WAIVS, the command-line decoder software is converted into a custom Netscape plug-in, named NpWvt.[2] The input arguments to the NpWvt plug-in (i.e. `inputFile, width, height, bitplaneOff, resolutionOff`) are provided as attributes in the `<embed>` tag , where they are specified using the form `name=value`, in an HTML document (see Section 4.3.6). The decoded images are displayed directly onto the Navigator browser window, instead of written out to files.

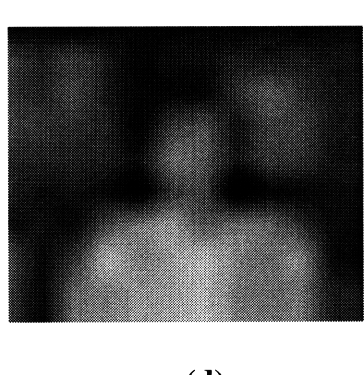
The NpWvt plug-in is created by modifying the source code of the template plug-in included in Netscape's plug-in software development kit (SDK) and combining it with the source code of the command-line version of the TI PEZW decoder software, with some modifications. Although the SDK is written in C++, the plug-in object model does

[2] An alternative approach would be to develop an ActiveX module to extend Microsoft Internet Explorer. The choice of Netscape plug-in over ActiveX module is due to the greater availability of literature, resources, and tools on developing Netscape plug-ins at the time of this implementation.

(a)

(b)

(c)



(d)

Figure 9: Reconstructed Akiyo Images for Various Decoder Settings:
(a) `bitplaneOff=0, resolutionOff=0` (original image);
(b) `bitplaneOff=0, resolutionOff=1;`
(c) `bitplaneOff=2, resolutionOff=0;`
(d) `bitplaneOff=7, resolutionOff=1`

not use the object-oriented features of the C++ language. Instead, plug-ins use an approach that could be implemented in C. The TI PEZW codec software is written entirely in C. The NpWvt plug-in is created for the Windows 95 platform, using the Win32 API. Visual C++ is used as the development environment for the plug-in.

The following sub-sections describe in detail the process of creating the NpWvt plug-in. Sections 4.3.1 and 4.3.2 give an overview of the Netscape plug-in life cycle and object model. Section 4.3.3 defines the NpWvt plug-in instance structure. Section 4.3.4 describes the implementation of some essential plug-in methods.

4.3.1 The Plug-in Life Cycle

A plug-in is a software module that can be added to Netscape Navigator to extend the browser's ability to display various types of data (e.g., audio, video, images, VR, PDF). In our case, the NpWvt plug-in enables Navigator to display images encoded with the TI PEZW codec. (For convenience, these images will hereafter be referred to as *WVT images.*)

Unlike standalone applications, a plug-in is not in charge of its own life cycle. Plug-ins are completely under the control of the Web browser and must be written to account for this architecture. For example, plug-ins are created automatically by Navigator when they are needed and can be destroyed at any time. The life cycle of a plug-in follows the pattern below:

- The user installs the plug-in.
- The user views a page that contains embedded data (WVT images in our case).

- Navigator identifies the appropriate plug-in to process and displays the data and loads the plug-in module, if needed.
- The Web browser creates a new instance of the plug-in.
- The browser provides the plug-in with a window in which the plug-in can display itself.
- The browser passes data to the plug-in.
- The plug-in processes and displays the data.
- When the user leaves the page, the browser destroys the plug-in instance.
- If there are no other instances of this plug-in, the browser unloads the plug-in module.

4.3.2 The Plug-in Object Model

Because of their dynamic life cycles, plug-ins are based on an object-oriented architecture. A plug-in is an *object* that supports a specific protocol that allows the object to interact with the Web browser. Each plug-in defines a data structure that maintains the plug-in's internal state. As with any object-oriented architecture, each *instance* of a plug-in has its own copy of this data, so multiple plug-ins of the same type can exist at once.

Plug-ins also have *methods*, which are functions that operate on the plug-in's instance data. There is a core set of methods, known as *plug-in methods*, that every plug-in must implement. These methods are called by Navigator at the appropriate time and are expected to perform some task required by the browser. For each plug-in there is a corresponding Navigator "object" that provides an interface between the plug-in and the browser. This object also has methods, which are known as *Netscape methods*. Netscape

methods provide various services that can be requested by plug-ins. Together, the plug-in methods and the Netscape methods provide the basis of all interactions between plug-ins and the browser, and enable the core capabilities of any plug-in.

4.3.3 The NpWvt Plug-in Instance Structure

The declaration of the data structure for the NpWvt plug-in object is shown in Listing 2. The data structure includes two variables to hold the input arguments `bitplaneOff` and `resolutionOff`, two variables to keep track of the dimensions of the decoded image, an array to store the RGB values of the decoded image, and other standard plug-in internal variables. An instance of this structure is passed to all plug-in methods called by Navigator.

```
typedef struct _PluginInstance {

    // Standard members for plug-ins

    NPWindow*      fWindow;    // pointer to plug-in window structure
    HWND           fhWnd;      // the plug-in window handle
    uint16         fMode;      // plug-in mode ("embedded" for NpWvt)
    FARPROC        fDefaultWindowProc;  // default subclass function

    // Members specific to the NpWvt plug-in

    int             bitplaneOff;
    int             resolutionOff;
    int             imageWidth;
    int             imageHeight;
    COLORREF *RGBbuffer;

} PluginInstance;
```

Listing 2: Declaration of the PluginInstance Data Structure

4.3.4 The NpWvt Plug-in Methods

The NpWvt plug-in implements methods that perform initialization when an instance is created and clean up when an instance is destroyed. Other methods respond to

incoming streams of data, and supply a visible user interface within a Web page. The functionality of the NpWvt plug-in is implemented by modifying the source code of the template plug-in found in the SDK. The only plug-in methods that need to be changed for the NpWvt plug-in are `NPP_New()`, `NPP_SetWindow()`, `NPP_NewStream()`, `NPP_Write()`, `NPP_DestroyStream()`, and `NPP_Destroy()`. These methods are discussed in the paragraphs that follow. All other plug-in methods remain unmodified. See [15] for a good description of standard plug-in methods.

When a plug-in is first loaded, Navigator calls the `NPP_New()` plug-in method to create a new instance of the plug-in. The NpWvt plug-in's `NPP_New()` method allocates memory for a copy of its `PluginInstance` structure, processes the argument list passed to the plug-in instance, and initializes the plug-in specific members of the instance data structure accordingly. Specifically, the `bitplaneOff` and `resolutionOff` members are set to their corresponding input arguments. `RGBbuffer` is set to the null pointer. The `imageWidth` and `imageHeight` members are not set at this point, since they refer to the final dimensions of the decoded image, which depends on `resolutionOff`.

Once a plug-in instance has been created, Navigator calls `NPP_SetWindow()` to provide a window in which the plug-in can display itself. This method may also be called if the window is resized, destroyed, or changed. The NpWvt plug-in's `NPP_SetWindow()` method performs some initial checks on the provided window. If the window has not been set previously, the method updates the `fWindow` and `fhWnd` members of the `PluginInstance` structure accordingly. This is followed by the installation of a *subclass function* for the window, a technique known as *subclassing* the

window. The subclass function, named WVT_SubClassFunc(), acts as an "event handler" for the plug-in window and must be implemented specifically for the NpWvt plug-in. The main purpose of this function is to redraw the screen whenever the NpWvt plug-in needs to be drawn. The implementation of WVT_SubClassFunc() will be described in Section 4.3.5. The default subclass function which has been replaced by WVT_SubClassFunc() is saved in the fDefaultWindowProc member of the PluginInstance structure.

Before the NpWvt plug-in has anything to draw to its given window, it must be able to read the WVT image data that Navigator provides. Navigator passes data to plug-ins in *streams*. When data is available, Navigator calls NPP_NewStream() to inform the plug-in that a new stream has been created and that data will follow. The NpWvt plug-in's NPP_NewStream() method implements data buffering by initializing a buffer structure defined in Listing 3.

```
typedef struct _PluginStream {
      unsigned char *data;      // buffered data
      int len;                  // length of buffer
} PluginStream;
```

Listing 3: Declaration of the PluginStream Buffer Structure

The PluginStream structure holds a character array and maintains the size of the buffer.

Once a stream has been opened, Navigator calls NPP_Write() as many times as necessary to transfer data to the plug-in. Since the data for an entire WVT image generally exceeds the amount of data that will arrive in a single call to NPP_Write() in real time, the incoming stream is buffered to ensure that the data is not split into unusable

segments. The buffer structure was already set up in `NPP_NewStream()`, as described earlier. The NpWvt plug-in's `NPP_Write()` method simply appends the new data to the buffer, extending the size of the buffer if necessary.

When all data has been transferred, Navigator calls `NPP_DestroyStream()` to inform the plug-in that the data transfer is complete and the stream is destroyed. At this point, the NpWvt plug-in is ready to decode the WVT image data it has collected in the stream buffer. It calls an external C function named `decode()`, which was previously the `main()` function in the command-line version of the TI PEZW decoder software. The `main()` function is modified to interface with the NpWvt plug-in. The new function prototype for the `decode()` function is shown in Listing 4.

```
void decode(unsigned char *buffer,
            int width, int height,
            int bitplaneOff, int resolutionOff,
            unsigned char *ImageData_R,
            unsigned char *ImageData_G,
            unsigned char *ImageData_B,
            int *imageWidth, int *imageHeight)
```

Listing 4: The Function Prototype for the `decode()` Function

Comparing this new syntax to the one shown in Section 4.2, we see that `inputFile` is replaced by the actual buffer of data which the plug-in provides. The `width`, `height`, `bitplaneOff`, and `resolutionOff` arguments remain the same. The new arguments `ImageData_R`, `ImageData_G`, and `ImageData_B` are references to three character arrays which the plug-in allocates in the `NPP_DestroyStream()` method, prior to calling `decode()`. The resulting RGB values for the decoded image will be stored into these arrays and passed back to the plug-

in. Similarly, references to the imageWidth and imageHeight members of the PluginInstance structure are passed to decode() for recording the actual dimensions of the decoded image.

Upon return from the decode() function, the NpWvt plug-in's NPP_DestroyStream() method packs the R, G, and B arrays into one RGB data structure, namely the RGBbuffer member of PluginInstance. Next, it frees memory allocated for the stream buffer and the R, G, and B arrays. Lastly, it calls the auxiliary functions InvalidateRect() and UpdateWindow() to force the plug-in window to be redrawn. This, in turn, causes a WM_PAINT message to be sent to the plug-in window, which will be handled by WVT_SubClassFunc(), where the actual displaying is performed (see Section 4.3.5).

When an instance of the NpWvt plug-in is destroyed, Navigator calls NPP_Destroy(). This method performs several cleanup steps. First, the original subclass function is restored to the plug-in window. Recall the handle to the default subclass function was stored in the PluginInstance structure in the NPP_SetWindow() method. Next, the method frees memory allocated for RGBbuffer. Finally, the instance structure itself is freed.

4.3.5 The Subclass Function

As mentioned earlier, the subclass function WVT_SubClassFunc() is automatically called whenever an event is received that should be handled by the plug-in window. The main purpose of this function is to redraw the screen whenever the NpWvt plug-in needs to be drawn. The function uses a switch statement to decide what action to

perform, based on the type of event message received from Windows. For example, the plug-in will receive a `WM_PAINT` message any time it needs to redraw itself. In `WVT_SubClassFunc()`, the `WM_PAINT` event is handled by calling the auxiliary function `SetPixel()` with the RGB values stored in `RGBbuffer`, which corresponds to the pixel values of the decoded image. As a result, the decoded image is displayed on the browser. The `imageWidth` and `imageHeight` members of `PluginInstance` are used to calculate offsets so that the decoded image is centered in the plug-in window.

4.3.6 MIME Types and The <embed> Tag

So far we have largely ignored what mechanism Navigator uses to determine which plug-in should be loaded for a given data type and, once an appropriate plug-in is found, how arguments are passed to the plug-in. Consider the sample HTML document in Listing 5.

```
<html>
<title> A Sample Web Page Which Calls The NpWvt Plug-in </title>
<embed src="http://web.mit.edu/simon888/www/akiyo.wvt"
       width=352 height=288 bitplaneOff=0 resolutionOff=1>
</embed>
</html>
```

Listing 5: A Sample Web Page Which Calls the NpWvt Plug-in

Every element on a Web page has an associated type. Every element's type is described using a simple protocol known as the "Multi-purpose Internet Mail Extensions" (MIME). The MIME type simply consists of a two-part string that specifies a major type and a subtype. For example, the MIME type that identifies a GIF image is "`image/gif`." For WVT images, we have defined the identifying MIME type to be "`application/x-waveletimage`."

When Navigator encounters an `<embed>` tag in an HTML document, it opens a connection to request data from the URL specified in the `src` attribute. The Web server associated with that URL looks at a MIME type registry to determine the type of data stored in the specified file. In general, the server determines MIME types by examining the file suffix. In this example, the server sees the .wvt file suffix and determines the data in the akiyo.wvt file is of the type `application/x-waveletimage`, assuming the server has been configured properly to recognize this MIME type. The server then replies to the Web browser by sending the MIME type for the data, followed by the data itself. Next, Navigator receives the MIME type and searches it's plug-in registry for a plug-in that can handle this data type. In this example, Navigator receives the `application/x-waveletimage` type and finds the NpWvt plug-in handles this MIME type, assuming the NpWvt plug-in was already installed and the plug-in registry has been configured properly to associate this MIME type to the plug-in. If a suitable plug-in is found, Navigator creates a plug-in instance and allocates an area in the page for the plug-in to use. Finally, Navigator passes the stream of data to the plug-in for it to handle as it sees fit.

An `<embed>` tag can have any number of attributes that may affect the operation or appearance of the plug-in. The `<embed>` tag in Listing 5 supplies `src`, `width`, and `height` attributes, which are recognized by all plug-ins. The NpWvt plug-in supports two additional attributes, `bitplaneOff` and `resolutionOff`, which are used as decoder settings, as described previously.

4.4 Server-Side Data Adaptation

The server portion of the system is a program that exploits the Common Gateway Interface (CGI) mechanism to run automatically when a Web client contacts the required URL. The CGI program serves as the "adaptive" portion of the system which controls how much data to send back to the Web client, based on some scalability parameters passed to it. The two components of the CGI program are:

- `nph-adapt.pl`: a Perl script which interfaces with the NpWvt plug-in
- `extract`: a C program which extracts the relevant layers of data from the WVT file

Section 4.4.1 presents the basics of CGI. Section 4.4.2 shows the implementation of `nph-adapt.pl`. Section 4.4.3 explains the WVT bitstream syntax and describes how the relevant layers of a WVT image file are extracted for delivery to the Web client.

4.4.1 CGI Basics

Before looking at the implementation of the server-side programs which perform the WVT data adaptation, it is important to understand some features of the Common Gateway Interface mechanism on which it is based. Clients (Web browsers) communicate with a Web server using a protocol known as HTTP. The HTTP protocol consists of a *request header* sent by the client to request some service and a corresponding *response header* sent by the server as its reply to the client.

Most servers support a special directory named cgi-bin, where CGI programs and scripts reside. When a user browses to a URL associated with a CGI program, the client sends an HTTP request to the server requesting the file. When servicing the request, the

server recognizes that the requested file is a program in the cgi-bin directory. Instead of redirecting the file back to the client verbatim, the server executes the CGI program. The program can read information sent by the browser, and can produce information that will be sent to the browser.

4.4.2 Perl Script

Although CGI programs can be written in any language, Perl, an interpreted scripting language, is the most popular one. Listing 6 shows a Perl implementation of server-side adaptation script, `nph-adapt.pl`.

```
#!/usr/local/bin/perl

print "HTTP/1.0 200\n";
print "Content-type: application/x-waveletimage\n\n";

$bitplaneOff = "0";
$resolutionOff = "0";

$ENV{'QUERY_STRING'} =~ s/%2C/,/g;
$ENV{'QUERY_STRING'} =~ s/=//g;

if ($ENV{'QUERY_STRING'}) {
   @params = split(/,/, $ENV{'QUERY_STRING'});
   ($bitplaneOff, $resolutionOff) = @params;
}

$output = `extract akiyo.wvt $bitplaneOff $resolutionOff`;
print $output;
```

Listing 6: The CGI Program, `nph-adapt.pl`

The above script is an example of a *non-parsed header (NPH)* CGI script. This means that the CGI script is responsible for generating and returning a complete HTTP response header. The alternative would be to generate only a partial response header and let the server intercept the output and complete the header information with header information of its own. Because non-parsed header CGI scripts return a complete

response header, no extra overhead is incurred by the server, and the output of the CGI script goes directly to the client. To identify themselves to the server as non-parsed header CGI scripts, programs that output complete headers must start with the "nph-" prefix. See [3] for a more detailed description of non-parsed header CGI programs.

The first two `print` statements correspond to the complete HTTP response header returned by `nph-adapt.pl`. The first header line is the status response header, which contains the HTTP protocol revision and the status of the program. The second header line is the Content-type response header, which indicates the MIME type of the data to follow. Note that a blank line after the last header line is required by the HTTP protocol to separate the header from the data that follows it.

Next, the `QUERY_STRING` environment variable is parsed for the `bitplaneOff` and `resolutionOff` arguments. The browser sends these values to the CGI script by appending them to the URL, after a question mark. The information appended to a URL after the question mark is known as a *query string*. When the server is passed a URL with a query string, it calls the CGI program identified in the first part of the URL (before the question mark) and stores the query string in the environment variable `QUERY_STRING`, which can be parsed by the CGI program for input arguments.

Finally, the script calls the `extract` program with the `bitplaneOff` and `resolutionOff` arguments and the name of the file containing the WVT data. The data returned from `extract` is sent directly to the browser.

4.4.3 The `extract` Program and the WVT Bitstream Syntax

The `extract` program is a standalone C program which generates on-the-fly a WVT bitstream to be sent to the browser by extracting the relevant layers of data from a WVT image file stored at the server. The size of the WVT bitstream is smaller than or equal to the size of the WVT file from which it is extracted. Yet the WVT bitstream contains all the information the NpWvt plug-in needs to decode the WVT image at the specified degradation level. Thus bandwidth is saved by eliminating the transfer of any data that is not needed for decoding a degraded version of the image. The calling syntax of `extract` is shown in Listing 7.

```
extract inputFile bitplaneOff resolutionOff
```

Listing 7: The Calling Syntax of `extract`

The `extract` program reads in the WVT image file specified by `inputFile` and parses it according to a defined WVT syntax (described in detail below). A subset of relevant data is determined from the specified values for the `bitplaneOff` and `resolutionOff` arguments. The header information and the subset of relevant data are copied to an output buffer, which is returned back to the calling Perl script. The extraction process is explored in greater detail later in this section.

In Figure 9, we saw qualitatively how the two decoder parameters `bitplaneOff` and `resolutionOff` affect the visual quality and size of the decoded image. We have not yet explored how the decoder exploits the embedded nature of the WVT image file format to generate these reconstructed images of various image qualities

and sizes. The remainder of this section examines the exact syntax for the WVT file format as well as the newly extracted WVT bitstream to be sent to the browser.

The TI PEZW codec supports both color and monochrome images. The syntax for a color WVT image file is given in the form of a pseudocode, as shown in Listing 8 below. Each line of the pseudocode contains the name of the field followed by the number of bits to represent that field in parenthesis.

```
wavelet_decomposition_levels (8 bits)
color (8 bits)
Y_mean (8 bits)
U_mean (8 bits)
V_mean (8 bits)
Quant_DC_Y (8 bits)
Quant_DC_UV (8 bits)
for (Y, U, V) {
   lowest_subband_bitstream_length (16 bits)
   lowest_subband_texture_coding (lowest_subband_bitstream_length bits)
}
spatial_scalability_levels (5 bits)
quantization_type (2 bits)
SNR_length_enable (1 bit)
for (Y, U, V) {
   for (i=0; i<spatial_scalability_levels; i++) {
      spatial_bitstream_length (24 bits)
      Quant (16 bits)
      for (j=0; j<SNR_scalability_levels; j++) {
         SNR_bitstream_length (16 bits)
         wavelet_zero_tree_coding (SNR_bitstream_length bits)
      }
   }
}
```

Listing 8: The WVT File Syntax

The syntax of a monochrome WVT file is very similar to the one shown above. The only difference is that any header fields and data layers associated with the U and V components of the image are absent from the monochrome WVT file syntax. Likewise, the process of extracting the relevant layers of data from a WVT file is almost identical for

both cases. The following explanation of the WVT file syntax and the bitstream extraction process will focus on the case of color WVT images only.

The first seven fields of the color WVT image file format are header information. The first field represents the number of decomposition levels, which corresponds to the total number of subband filtering stages in the encoding (as discussed in Chapter 3). The second field is a flag that indicates whether the image is color or monochrome. The next five fields specify the mean value for each of the Y, U, and V components of the image, the quantizer for the DC layer (lowest subband) of the Y component, and the common quantizer for the DC layers (lowest subbands) for the U and V components.

Next, the data corresponding to the DC layer for each of the Y, U, and V components is followed. Each DC layer is preceded by a 16-bit header field which specifies the length of the data.

Three more header fields come after the DC layers. They specify the number of spatial scalability layers, the quantization type, and the SNR bitstream length mode toggle. All fields up to this point must be included in the extracted WVT bitstream for any degradation levels (i.e. for any values of `bitplaneOff` and `resolutionOff`).

The remaining fields consist of the header and data for each spatial and SNR scalability layer. Note that the data associated with the Y, U, and V components are not interleaved in these layers of data. All layers of Y come first, followed by all layers of U, followed by all layers of V.

Each spatial scalability layer begins with two header fields that specify the size of the bitstream and the quantizer for the current spatial scalability layer. The bitstream for this spatial layer follows, which is in turn divided into SNR scalability layers. Each SNR

scalability layer begins with a 16-bit field that indicates the size of the current SNR scalability layer, followed by the bitstream for this SNR layer itself.

The `resolutionOff` parameter tells the `extract` program how many spatial layers to skip when extracting the WVT bitstream from the WVT image file. Likewise, `bitplaneOff` controls the number of SNR layers skipped in every spatial layer. Listing 9 shows the resulting syntax for the newly extracted WVT bitstream.

```
wavelet_decomposition_levels (8 bits)
color (8 bits)
Y_mean (8 bits)
U_mean (8 bits)
V_mean (8 bits)
Quant_DC_Y (8 bits)
Quant_DC_UV (8 bits)
for (Y, U, V) {
   lowest_subband_bitstream_length (16 bits)
   lowest_subband_texture_coding (lowest_subband_bitstream_length bits)
}
spatial_scalability_levels (5 bits)
quantization_type (2 bits)
SNR_length_enable (1 bit)
for (Y, U, V) {
   for (i=resolutionOff; i<spatial_scalability_levels; i++) {
      spatial_bitstream_length (24 bits)
      Quant (16 bits)
      for (j=0; j<SNR_scalability_levels-bitplaneOff; j++) {
         SNR_bitstream_length (16 bits)
         wavelet_zero_tree_coding (SNR_bitstream_length bits)
      }
   }
}
```

Listing 9: The Syntax for the Extracted WVT Bitstream

The NpWvt plug-in must be modified accordingly to support this new syntax. Otherwise, the decoding process will fail.

4.5 Sample Execution of the System

Putting all the pieces together, the following sample HTML document, shown in Listing 10, demonstrates how the adaptive wavelet-based image viewing system works.

```
<html>
<title> A Sample Web Page Which Calls The NpWvt Plug-in </title>

<script LANGUAGE="JavaScript">

var largeExpDate = new Date ();
var myDomain;

.
.
.

function SetParams()
{
  var bitplaneOff = GetCookie('bitplaneOff');
  var resolutionOff = GetCookie('resolutionOff');

  if ((bitplaneOff == null) || (resolutionOff == null))
  {
    bitplaneOff=0;
    resolutionOff=0;
  }

  document.write('<center>');
  document.write('<embed src="http://web.mit.edu/simon888/cgi-bin/
  nph-adapt.pl?' + bitplaneOff + ',' + resolutionOff + '"');
  document.write(' width=352 height=288');
  document.write(' bitplaneOff=' + bitplaneOff);
  document.write(' resolutionOff=' + resolutionOff);
  document.write('></center>');

}
SetParams();

</script>

</html>
```

Listing 10: A Sample Web Page Which Illustrates the WAIVS At Work

Unlike the sample Web page shown in Listing 5 (Section 4.3.6), this example uses javascript to generate various attributes in the <embed> tag. This example assumes that values for the `bitplaneOff` and `resolutionOff` variables are previously stored in a

client-side cookie. Each time this sample Web page is loaded, the javascript function `SetParams()` fetches these values from a cookie by calling `GetCookie()`, another javascript function not shown here (see the Appendix, Section 9.5). The corresponding attributes of the `<embed>` tag are properly set, and a query string consisting of these values is appended to the URL specified in the `src` attribute. Also, the URL in the `src` attribute now references the CGI program "`nph-adapt.pl`" instead of a static WVT file, as in the previous example.

When a user browses this sample Web page, the following sequence of actions occurs:

1. The browser sends an HTTP request to the server for the sample Web page.
2. The server sends an HTTP response header along with the HTML document requested back to the client.
3. At the client side, javascript fetches the values of `bitplaneOff` and `resolutionOff` from a cookie and replaces any references to these variables in the HTML document (e.g. in the "query string" field of the `src` attribute, and in the `bitplaneOff` and `resolutionOff` attributes of the `<embed>` tag) with the fetched values.
4. The browser encounters the `<embed>` tag and sends an HTTP request to the server associated with the URL specified in the `src` attribute.
5. Since the URL corresponds to a CGI program, the server executes the CGI program, instead of sending an HTTP response header along with the CGI program back to the client verbatim. The query string appended at the end of the URL is stored in the environment variable `QUERY_STRING`.

6. The CGI program uses the information in `QUERY_STRING` to set up a call to the `extract` program.
7. The `extract` program extracts the appropriate layers of data from the requested WVT file to form the WVT bitstream, which is returned to the CGI program.
8. Since it is "non-parsed," the CGI program generates the appropriate HTTP response header (which includes information about the content MIME type, `application/x-waveletimage`) and sends it along with the WVT bitstream back to the client directly, without going through the server.
9. The browser finds the appropriate plug-in for the MIME type of the incoming data and loads it. A new instance of the NpWvt plug-in is created. The values of the `bitplaneOff` and `resolutionOff` attributes in the `<embed>` tag are passed to the NpWvt plug-in. The WVT bitstream is re-directed to the plug-in as well.
10. The NpWvt plug-in buffers the data, decodes it, and displays the decoded image on the browser window.

5. System Evaluation

5.1 Time Efficiency

In evaluating the time efficiency of WAIVS, we examine the total latency between the initial request for an image and the display of the decoded image on the browser, which roughly consists of the following:

1. Server response time

2. Adaptation time: time to execute `extract`, the adaptive degradation module.

3. Network latency

4. Connection latency: time to transfer the incoming bitstream across the client's network connection, typically a modem or an Ethernet card.

5. Decode time: time to decode the WVT bitstream.

Server response time and network latency are difficult to quantify, as they vary greatly depending on numerous dynamic factors and the capabilities of each server and network. Hence we will focus the evaluation on the adaptation time, connection latency, and decode time.

The execution time of the adaptive degradation module contributes a very small portion of the total latency because of its simple operation. For typical server loads and WVT image file sizes (2 to 10 kilobytes), `extract` should take no longer than a quarter of a second to degrade one WVT image. The exact execution time depends on the server load and may increase with the number of instances of `extract` running concurrently.

The connection latency depends on the size of the incoming WVT bitstream and the connection speed. Figure 10 shows the reduction in connection latency when the

degraded versions, instead of the original version, of the Akiyo test image are downloaded across modems running at 28.8, 33.6, and 56 kilobits per second.

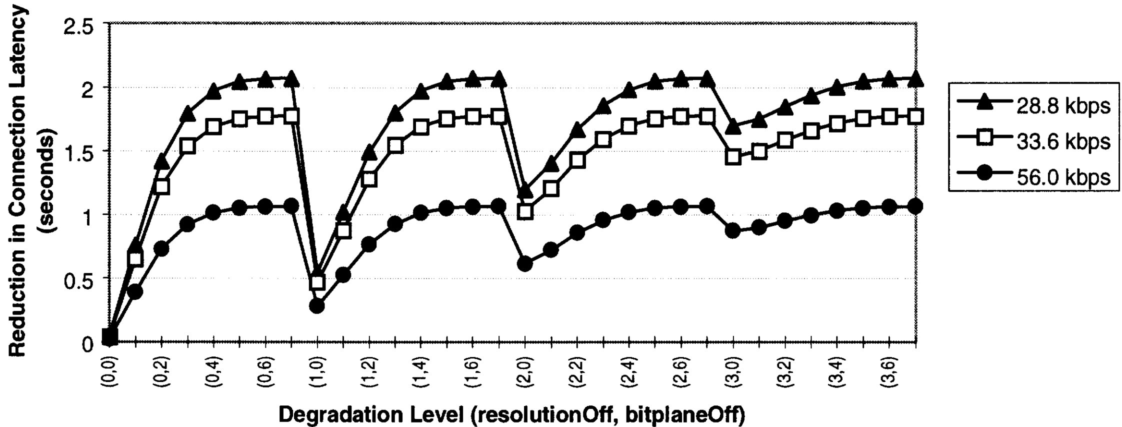


Figure 10: Reduction in Connection Latency Versus Degradation Level for the Akiyo Test Image (352x288 pixels, 63752 original bitstream size)

Note that the reduction in connection latency is quite significant, ranging from about 0.5 to 2 seconds.

Still in the early stage of development, the focus of the TI PEZW decoder remains coding efficiency rather than speed. Hence the current version of the PEZW decoder takes a long time to decode a WVT image. The decode time typically ranges from 0.5 second to 15 seconds, depending on the size and degradation level of the WVT image. We believe that the C implementation of the PEZW decoder can be optimized to achieve decoding times that are at least comparable to JPEG.

5.2 Server Load Issues

In WAIVS, each request for a WVT image generates an instance of the CGI program. If many requests come at the same time, which is possible during peak hours

when many people are browsing Web pages containing inline WVT images, the many CGI instances running concurrently place a heavy burden on the server. In dealing with the aforementioned problem with server load, WAIVS can be extended to provide an option to bypass the adaptive degradation module. Web page designers can also be advised to reference the URLs corresponding to the WVT images directly, instead of the CGI programs, so that calls to the CGI programs are bypassed altogether. For both approaches, the adaptivity of network bandwidth is lost. However, the adaptation hints can still be fed to the NpWvt plug-in so that the adaptivity of processing (decoding) time is retained. This is illustrated in the HTML example in Listing 11.

```
<html>
<title> A Sample Web Page Which Calls The NpWvt Plug-in </title>

<script LANGUAGE="JavaScript">

var largeExpDate = new Date ();
var myDomain;

.
.
.

function SetParams()
{
  var bitplaneOff = GetCookie('bitplaneOff');
  var resolutionOff = GetCookie('resolutionOff');

  if ((bitplaneOff == null) || (resolutionOff == null))
  {
    bitplaneOff=0;
    resolutionOff=0;
  }

  document.write('<center>');
  document.write('<embed src="http://web.mit.edu/akiyo.wvt"');
  document.write(' width=352 height=288');
  document.write(' bitplaneOff=' + bitplaneOff);
  document.write(' resolutionOff=' + resolutionOff);
  document.write('></center>');

}
SetParams();

</script>
```

```
</html>
```

Listing 11: Sample Web Page Illustrating How To Bypass Adaptive Degradation While Retaining Adaptivity In Decoding Time.

5.3 Benefits of Adaptive Degradation

5.3.1 Better Viewing Experience for the Users

WAIVS is beneficial to the people who browse images on the Web because it offers a new dimension of choices, namely, images at different information resolutions. Adaptivity offers more timely delivery of the images across a wide range of client and network capabilities. The user is given the choice to download a lower-quality picture instead of waiting a long time for a full-quality picture to download. Moreover, since the user can specify the adaptation hints, users can exert some control over the degradation level of the downloaded images.

5.3.2 Ease of Use for Content Creators

An adaptive image viewing system simplifies the task of creating Web pages that are viewable for a wide range of network and client capabilities. For example, two different versions (one for low-bandwidth and another for high-bandwidth) of Web pages are not needed, since WAIVS automatically generates the different versions. WAIVS also provides content creators the flexibility to be more liberal in the use of images in Web pages, since the system "takes care" to adaptively degrade the images so the download time for low-bandwidth clients is acceptable.

Encoding images into the WVT format is just as easy as encoding them into other formats such as JPEG and GIF. The only additional work is to provide the PEZW encoder with some parameters, such as the number of decomposition levels, and spatial and SNR scalability levels, which is trivial. By specifying these parameters to the PEZW encoder, the content creators have full control of the degradation behavior of the images created. To achieve the desired look of the images at different degradation levels, the content creators can iteratively try out several sets of encoder parameters and view the resulting pictures at various degradation levels using the browser plug-in. Thus WAIVS provides many benefits to the content creators with very little work.

5.3.3 Elimination of Wasted Bandwidth

Precious network bandwidth is wasted in the mismatch of interest between the content creator and the user. For example, suppose the author of a web page included a 80-Kbyte, good-quality image of a man in a tuxedo standing next to a penguin, but a visitor to this page only cared about the relative sizes of the two (and missed the humor too). Then a 20-Kbyte, intelligible picture would have sufficed and the difference of 60 Kbytes would have been saved. More generally speaking, whenever an image contains a higher level of detail than a viewer cares to observe, some bandwidth is wasted in transmitting the extra information which is irrelevant to that particular viewer. The degradation scheme in WAIVS is based on adaptation hints which account for user interest. As a result, wasted bandwidth due to sending unnecessary image details to the user is eliminated.

6. Conclusion

This research is in response to the rigidity of image formats and conventions that view images as static objects in terms of size and quality. Here we present an alternative image representation which allows for the adaptive degradation of the picture, enabling the efficient use of network bandwidth and timely delivery of images over a wide variety of technologies.

An adaptive image degradation model is proposed and a wavelet-based adaptive image viewing system (WAIVS) is developed. The adaptive image degradation model achieves adaptive data rate for a given image without extra storage requirement at the server. The cost comes in the added complexity in the adaptive degradation process, each time an image is requested. In WAIVS, the highly scalable and embedded image representation provided by wavelet-based image coding technology is exploited to minimize the overhead cost of adaptively degrading an image on-the-fly.

WAIVS is built on the Web's client-server model. The server portion of WAIVS is a CGI program that is responsible for the adaptive degradation of images based on some adaptation hints provided by the client. The client portion of the system consists of a browser plug-in, built from the TI PEZW decoder, to decode WVT images, and some javascripts to look up and deliver adaptation hints to the server.

WAIVS offers numerous advantages over existing means of viewing images over the Web, including higher compression ratios, timely delivery of images to low-bandwidth clients, flexibility in designing graphics intensive Web pages, and more efficient use of network bandwidth.

7. Further Work

7.1 Per Object Adaptive Degradation

Instead of degrading the entire image uniformly, in some cases, it may be desirable to divide the image into regions representing different *objects* and degrade each region separately to a varying extent. This capability is known as *content-based scalability* in the upcoming MPEG4 standard [2] [10]. For example, the content creator may wish to retain the quality of the foreground objects as much as possible. As shown in the sequence of images in Figure 11, as `bitplaneOff` increases, the two cars in the foreground degrade at a slower rate than the background.

Figure 11: Sample images illustrating per object adaptive degradation:

`bitplaneOff` = 0 (top left), 3 (top right), 5 (bottom left), and (bottom right)

In extending the adaptive image degradation model introduced in this paper to support per object adaptive degradation, some interesting questions arise. What are the heuristics of translating a single set of adaptation hints for the whole image into different sets of adaptation hints for the different objects within the image? How are the sub-images stored on the server? Are they stored as separate WVT files? Are the degraded versions of the sub-images transmitted as separate bitstreams to the client? If so, how are they decoded and combined into a single image to be displayed on the browser? Can arbitrary-shaped objects be supported? Do the benefits of per object adaptive degradation outweigh the implementation complexity and the overhead costs during execution? The MPEG4 standard aims to provide an "object layered" bitstream format in which each audio or video object is coded into a separate object bitstream layer. This coded representation of audio and video data allows the user the separate decoding and reconstructions of the objects at different quality levels [2] [10].

7.2 Progressive Scanning

The idea of progressive scanning is borrowed from the progressive JPEG standard. The incoming bitstream is decoded on-the-fly as it arrives at the browser and the decoded image is displayed in a series of scans. Thus the user can see an approximation of the whole image very quickly, with gradual improvement of quality as one waits longer.

One way to extend WAIVS to support progressive scanning is illustrated in Figure 12. An additional parameter, `proScanEnable`, is sent from the client to the server to toggle between progressive scanning mode and standard mode (without progress

scanning). In progressive scanning mode, the adaptation hints are ignored at the server and the entire original WVT image is sent to the client without any degradation. At the client, `bitplaneOff` and `resolutionOff` are provided to the NpWvt plug-in so the WVT decoder can start decoding as soon as enough data has arrived to reconstruct an image at the degradation level specified by the adaptation hints. That is, the image displayed for the first scan in progressive scanning mode is the same as the degraded image in standard mode. Subsequent scans are produced by decoding more and more data as it arrives. Each scan adds to the data already provided, so that the total storage requirement is no greater than if the image is decoded and displayed after the entire bitstream is downloaded. The last scan corresponds to the original image at full quality.

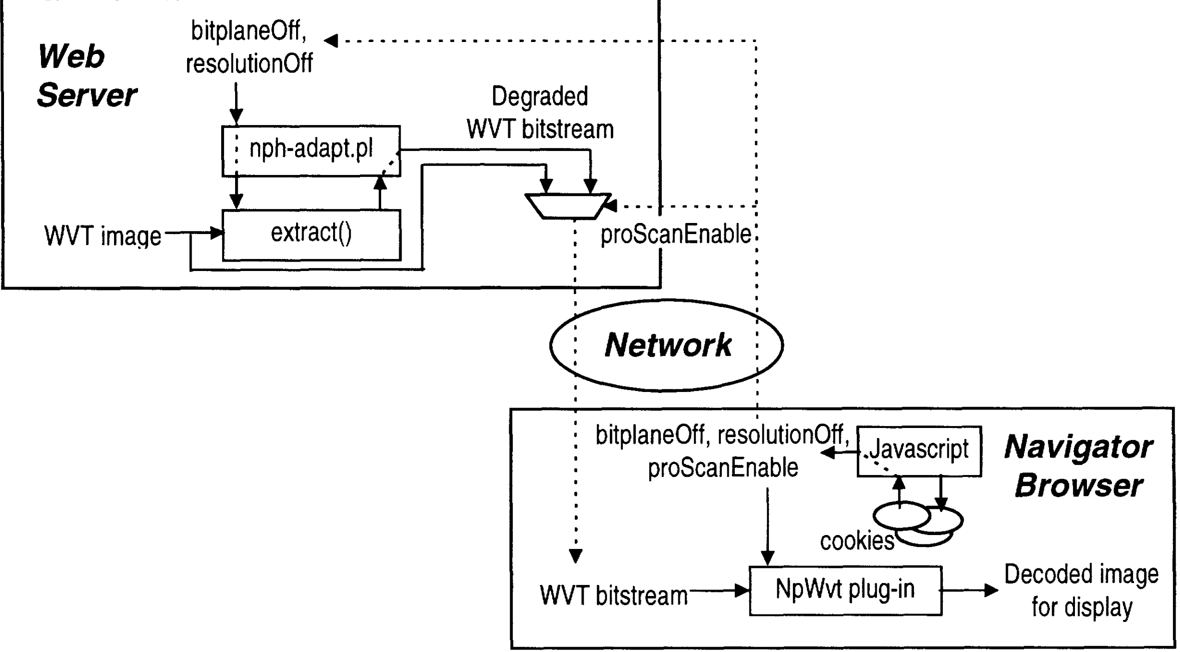


Figure 12: Extending WAIVS to support progressive scanning.

The advantage of progressive scanning is the user can see both a preview of the image very quickly and the full quality image shortly after. The first scan would probably

come faster than the degraded image in standard mode because the adaptive degradation process at the server is bypassed in progressive scanning mode. The downside of progressive scanning is there is no saving in network bandwidth. Also, multiple scans takes more computation time than decoding the entire WVT bitstream once it is fully downloaded. Therefore, progressive scanning only makes sense if one has a decoder which is fast compared to the communication link.

7.3 Extension to Video

Using the framework suggested in this work, the adaptive degradation model can be extended to other types of media. Adaptive degradation techniques for video would unquestionably be highly valuable, since video clips require considerably greater bandwidth than images. Frame rate is a reasonable adaptive characteristic. With temporal scalability, a video can be degraded on-the-fly by dropping frames. Hence the data rate is reduced at the cost of the video appearing more jerky.

The major difficulty of extending the aforementioned adaptive degradation model to video lies in designing an embedded video bitstream that supports spatial, SNR, and temporal scalabilities simultaneously. Moreover, the desirability for real-time presentation of video further constrains the complexity of the encoding format. Scalable video coding remains an area of active research.

8. Bibliography

[1] V. M. Bove, Jr., "Scalable (Extensible, Interoperable) Digital Video Representations," in Digital Images and Human Vision, A. B. Watson, ed., MIT Press, Cambridge MA, 1993, pp. 23-34.

[2] Leonardo Chiariglione, "MPEG and Multimedia Communications", *IEEE Trans*, CSVT, Vol. 7, No. 1, February 1997.

[3] Shishir Gundavaram, *CGI Programming on the World Wide Web.* O'Reilly, CA, 1996.

[4] John C. Huffman, "Wavelets and Image Compression," *SMPTE Journal*, pp. 723-727, November 1994.

[5] E. Lamboray, "Progressive Transmission For Very Low Bit Rate Context Based Image Compression," Technical Report CAIP-TR 218, Rutgers University, NJ, June 1997.

[6] Jie Liang, "Highly Scalable Image Coding for Multimedia Applications," *Proceedings of the 5th ACM International Conference on Multimedia*, Seattle, WA, November 1997.

[7] Karen Oehler, "Low Bitrate Wavelet Video Compression Scheme," Texas Instruments Technical Activity Report, December 1994.

[8] Karen Oehler, "Low Bitrate Wavelet Video Compression Scheme, Part 2," Texas Instruments Technical Activity Report, May 1995.

[9] Jerome M. Shapiro, "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Transactions on Signal Processing*, Vol. 41, No. 12, pp. 3445-3462, December 1993.

[10] T. Sikora, "The MPEG-4 Video Standard Verification Model," *IEEE Trans*, CSVT, Vol. 7, No. 1, February 1997.

[11] Gilbert Strang, "Wavelets and Dilation Equations: A Brief Introduction," *SIAM Review*, Vol. 31, No. 4, pp. 614-627, December 1989.

[12] Gilbert Strang, "Wavelets," *American Scientist*, Vol. 82, pp. 250-255, May-June 1994.

[13] Larry Wall, Tom Christiansen, and Randal L. Schwart, *Programming Perl.* O'Reilly, CA, 1996.

[14] Andrew Wooldridge, Mike Morgan, Mark C. Reynolds, and Jerry Honeycutt, *Special Edition Using JavaScript, Second Edition.* Que, IN, 1997.

[15] Douglas A. Young, *Netscape Developer's Guide to Plug-ins.* Prentice Hall, NJ, 1997.

9. Appendix

9.1 Comparison of Encoding Quality between Wavelets and JPEG

The following three sets of images show the superior quality of wavelet-encoded images as compared to JPEG images at comparable bitrates. Although shown as black-and-white images in the following figures, the images used for this comparison are color images. The wavelet-encoded images are generated from the TI PEZW codec at three quality-level settings, which are `bitplaneOff=0`, `bitplaneOff=1`, and `bitplaneOff=2`. The corresponding JPEG images are generated from the standard JPEG encoder using appropriate quality-level settings, called q-factors. The appropriate q-factors are determined by trial-and-error to match the bitrates of the three wavelet-encoded images.

Figure 13: Comparison of Wavelet-encoded Image and JPEG Image at Comparable Bitrates: 62392 bits (Wavelet, top), 62240 bits (JPEG, bottom)

Figure 14: Comparison of Wavelet-encoded Image and JPEG Image at Comparable Bitrates: 41296 bits (Wavelet, top), 41024 bits (JPEG, bottom)

Figure 15: Comparison of Wavelet-encoded Image and JPEG Image at Comparable Bitrates: 21904 bits (Wavelet, top), 21784 bits (JPEG, bottom)

9.2 NpWvt.cpp

```
#include "windows.h"
#include "string.h"
#include <ctype.h>
#include "npapi.h"
extern "C" {
    #include "decode.h"
}

typedef struct _PluginStream {
    unsigned char *data;
    int len;
} PluginStream;

typedef struct _PluginInstance {

    // Standard entries for plug-ins

    NPWindow*       fWindow;
    HWND            fhWnd;
    uint16          fMode;
    FARPROC         fDefaultWindowProc;

    // Members specific to the NpWvt plug-in

    COLORREF        *RGBbuffer;
    int             bitplaneOff;
    int             resolutionOff;
    int             imageWidth;
    int             imageHeight;

} PluginInstance;

LONG NP_LOADDS WINAPI
WVT_SubClassFunc(HWND hWnd,WORD Message,WORD wParam, LONG lParam);


NPError NPP_Initialize(void) {
    return NPERR_NO_ERROR;
}


void NPP_Shutdown(void) {
}


NPError NPP_New ( NPMIMEType   pluginType,
                  NPP          instance,
                  uint16       mode,
                  int16        argc,
                  char        *argn[],
                  char        *argv[],
                  NPSavedData *saved )
{
    // A NULL instance pointer would indicate a
    // serious error.

    if ( instance == NULL )
        return NPERR_INVALID_INSTANCE_ERROR;

    // Allocate memory for the PluginInstance structure.
    // For convenience, assign the memory a local pointer
    // named "This"
```

```
    instance->pdata = NPN_MemAlloc(sizeof(PluginInstance));
    PluginInstance* This = (PluginInstance*) instance->pdata;

    // Test for memory failure

    if ( This == NULL)
        return NPERR_OUT_OF_MEMORY_ERROR;

    // Initialize the PluginInstance structure

    This->fWindow = NULL;
    This->fMode   = mode;
    This->fhWnd   = NULL;

    // Initialize the data specific to this plugin

    This->RGBbuffer = NULL;

    // Compare each entry in the argn array for arguments
    // this plugin supports.

    for (int i = 0; i < argc; i++ )
    {
        if ( !::stricmp ( argn[i], "bitplaneOff" ) )
           This->bitplaneOff = ::atoi( argv[i] );
        else if ( !::stricmp ( argn[i], "resolutionOff" ) )
           This->resolutionOff = ::atoi( argv[i] );
    }

    return NPERR_NO_ERROR;
}


NPError NPP_Destroy ( NPP instance, NPSavedData** save )
{
    if ( instance == NULL )
        return NPERR_INVALID_INSTANCE_ERROR;

    PluginInstance* This = (PluginInstance*) instance->pdata;

    if (This != NULL)
    {
        // Remove the subclass function
        // for the client window

        if (This->fhWnd)
            ::SetWindowLong(This->fhWnd, GWL_WNDPROC,
                            (DWORD)This->fDefaultWindowProc);

        // Free the memory allocated in NPP_New()

        if (This->RGBbuffer)
            NPN_MemFree(This->RGBbuffer);

        NPN_MemFree(instance->pdata);
        instance->pdata = NULL;
    }

    return NPERR_NO_ERROR;
}


#define PROPERTY "PluginInstance"

NPError
NPP_SetWindow(NPP instance, NPWindow* window)
```

```
{
    if (instance == NULL)
        return NPERR_INVALID_INSTANCE_ERROR;

    PluginInstance* This = (PluginInstance*) instance->pdata;

    if( This->fWindow != NULL )  /* If we already have a window, clean
                                  * it up before trying to subclass
                                  * the new window. */
    {
        if( (window == NULL) || ( window->window == NULL ) )
        {
            /* There is now no window to use. get rid of the old
             * one and exit. */
            ::SetWindowLong( This->fhWnd, GWL_WNDPROC,
                             (LONG)This->fDefaultWindowProc);
            This->fDefaultWindowProc = NULL;
            This->fhWnd = NULL;
            This->fWindow = window;
            return NPERR_NO_ERROR;
        }
        else
            if ( This->fhWnd == window->window )
            {
                /* The new window is same as the old one. Redraw and get out.
*/
                ::InvalidateRect( This->fhWnd, NULL, TRUE );
                ::UpdateWindow( This->fhWnd );
                This->fWindow = window;
                return NPERR_NO_ERROR;
            }
            else
            {
                /* Clean up the old window, so that we can subclass the new
                 * one later. */
                ::SetWindowLong( This->fhWnd, GWL_WNDPROC,
                                 (LONG)This->fDefaultWindowProc);
                This->fDefaultWindowProc = NULL;
                This->fhWnd = NULL;
            }
    }
    else
        if( (window == NULL) || ( window->window == NULL ) )
        {
            /* We can just get out of here if there is no current
             * window and there is no new window to use. */
            This->fWindow = window;
            return NPERR_NO_ERROR;
        }

    /* At this point, we will subclass
     * window->window so that we can begin drawing and
     * receiving window messages. */

    This->fWindow = window;
    This->fhWnd   = (HWND)(DWORD)This->fWindow->window;

    // Subclass the window instance, saving the
    // previous WNDPROC in the PluginInstance
    // record.

    This->fDefaultWindowProc =
        (FARPROC) ::SetWindowLong ( This->fhWnd,
                                    GWL_WNDPROC,
                                    (DWORD) WVT_SubClassFunc );

    // Store this plugin's PluginInstance as a
```

```
    // property associated with the Window handle.

    ::SetProp ( This->fhWnd, PROPERTY, This );

    return NPERR_NO_ERROR;
}


NPError NPP_NewStream ( NPP         instance,
                        NPMIMEType type,
                        NPStream   *stream,
                        NPBool      seekable,
                        uint16     *stype )
{
    PluginStream* newStream;

    if (instance == NULL)
        return NPERR_INVALID_INSTANCE_ERROR;

    PluginInstance* This = (PluginInstance*) instance->pdata;

    // Create a stream data instance, and store it in the NPStream

    newStream = (PluginStream*) NPN_MemAlloc(sizeof(PluginStream));

    if (newStream != NULL)
    {
        newStream->data = NULL;
        newStream->len = 0;
        stream->pdata = newStream;
    }
    else {
        return NPERR_OUT_OF_MEMORY_ERROR;
    }

    return NPERR_NO_ERROR;
}


#define DATA_SIZE 0X0FFFFFFF

int32 NPP_WriteReady ( NPP instance, NPStream *stream )
{
    // Return the number of bytes this plugin is
    // ready to accept in the next call to NPP_Write()

    return DATA_SIZE;
}

int32 NPP_Write ( NPP instance, NPStream *stream,
                  int32 offset, int32 len, void *buffer )
{
    if ( instance != NULL )
    {
        // If the existing stream already has some data,
        // copy the data into a temporary stream structure,
        // append it with the incoming data from buffer,
        // and copy the appended stream back to the old stream

        if (stream->pdata != NULL)
        {
            PluginStream* ps = (PluginStream*) stream->pdata;

            if (ps->data == NULL)
            {
                ps->data = (unsigned char*) NPN_MemAlloc(len);
                ps->len = 0;
```

```
            }
            else
            {
                // Create a larger buffer for the stream
                unsigned char *data =
                    (unsigned char*) NPN_MemAlloc(len+ps->len);

                // Copy old data into new stream buffer
                ::memcpy(data, ps->data, ps->len);

                // Deallocate old stream buffer
                NPN_MemFree(ps->data);

                // Update pointer to point to new stream buffer
                ps->data = data;
            }

            // Append new data into new stream buffer
            ::memcpy (&ps->data[ps->len], (unsigned char*) buffer, len);
            ps->len += len;

            return len;
        }
    }

    // A NULL instance would constitute an error

    return -1;
}

NPError  NPP_DestroyStream ( NPP       instance,
                             NPStream *stream,
                             NPError   reason )
{
    if ( instance == NULL )
        return NPERR_INVALID_INSTANCE_ERROR;

    if (reason == NPRES_DONE)
    {
        PluginInstance* This = (PluginInstance*) instance->pdata;
        PluginStream* ps = (PluginStream*) stream->pdata;

        if (ps != NULL)
        {
            if (ps->data != NULL)
            {
                // allocate three buffers for R, G, and B values

                int bufferSize = This->fWindow->height*This->fWindow->width;
                unsigned char *R = (unsigned char *) NPN_MemAlloc(bufferSize);
                unsigned char *G = (unsigned char *) NPN_MemAlloc(bufferSize);
                unsigned char *B = (unsigned char *) NPN_MemAlloc(bufferSize);

                NPN_Status(instance, "Decoding image ...");

                decode (ps->data,
                        This->fWindow->width,
                        This->fWindow->height,
                        This->bitplaneOff,
                        This->resolutionOff,
                        R, G, B,
                        &This->imageWidth,
                        &This->imageHeight);

                // free up memory allocated for stream

                NPN_MemFree(ps->data);
```

```
                // compact into one buffer of RGB (COLORREF) values

                This->RGBbuffer =
                    (COLORREF *) NPN_MemAlloc(bufferSize*sizeof(COLORREF));

                for (int i=0; i<bufferSize; i++)
                    This->RGBbuffer[i] = RGB(R[i],G[i],B[i]);

                // free up memory allocated for R, G, B buffers

                NPN_MemFree(R);
                NPN_MemFree(G);
                NPN_MemFree(B);

                // force a window repaint
                NPN_Status(instance, "Decoding image ... done");

                ::InvalidateRect ( This->fhWnd, NULL, TRUE );
                ::UpdateWindow ( This->fhWnd );
            }
            NPN_MemFree(ps);
            stream->pdata = NULL;
        }
    }
    return NPERR_NO_ERROR;
}


void NPP_StreamAsFile(NPP instance, NPStream *stream, const char* fname)
{
    if (instance == NULL)
        return;

    PluginInstance* This = (PluginInstance*) instance->pdata;
}


void NPP_URLNotify(NPP instance, const char* url,
                   NPReason reason, void* notifyData)
{
}


void NPP_Print ( NPP instance, NPPrint* printInfo )
{
    if ( printInfo == NULL )
        return;

    if ( instance != NULL )
    {
        PluginInstance* This =
           (PluginInstance*) instance->pdata;

        if (printInfo->mode == NP_FULL)
        {
            NPFullPrint *fp;

            fp = &(printInfo->print.fullPrint);

            fp->pluginPrinted = FALSE;
        }
        else   // If not fullscreen, plugin must be embedded
        {
        }
    }
}
```

```
int16 NPP_HandleEvent(NPP instance, void* event)
{
    NPBool eventHandled = FALSE;
    if (instance == NULL)
        return eventHandled;

    PluginInstance* This = (PluginInstance*) instance->pdata;

    return eventHandled;
}


LONG WINAPI WVT_SubClassFunc ( HWND hWnd,
                               WORD Message,
                               WORD wParam,
                               LONG lParam )
{
    // Get the plugin instance associated with this window

    PluginInstance *This =
        (PluginInstance *) ::GetProp(hWnd, PROPERTY);

    switch (Message)
    {
        case WM_PAINT:
        {
            PAINTSTRUCT paint;

            // All paint calls must be bracketed
            // by ::BeginPaint() and ::EndPaint()

            HDC hDC = ::BeginPaint(hWnd, &paint);

            // Calculate offsets to center the image
            int xOffset, yOffset;
            xOffset = (This->fWindow->width - This->imageWidth)/2;
            yOffset = (This->fWindow->height - This->imageHeight)/2;

            // Display image buffer
            COLORREF *RGB = This->RGBbuffer;

            if (RGB != NULL)
                for (int y=0; y<(int)This->imageHeight; y++)
                    for (int x=0; x<(int)This->imageWidth; x++)
                        ::SetPixelV(hDC, x+xOffset, y+yOffset, *RGB++);

            // Call ::EndPaint() after all drawing has been completed
            ::EndPaint ( hWnd, &paint );
        }
        return 0;
        break;

        default:
            // For all other messages, call the original WndProc
            return ::CallWindowProc(This->fDefaultWindowProc,
                                    hWnd, Message, wParam, lParam );
    }
}


jref NPP_GetJavaClass(void)
{
    return NULL;
}
```

9.3 nph-adapt.pl

```
#!/map/sys/bin/perl

print "HTTP/1.0 200\n";
print "Content-type: application/x-waveletimage\n\n";

$bitplaneOff = "0";
$resolutionOff = "0";

$ENV{'QUERY_STRING'} =~ s/%2C/,/g;
$ENV{'QUERY_STRING'} =~ s/=//g;

if ($ENV{'QUERY_STRING'}) {
   @params = split(/,/, $ENV{'QUERY_STRING'});
   ($bitplaneOff, $resolutionOff) = @params;
}

$output = `extract akiyo.wvt $bitplaneOff $resolutionOff`;
print $output;
```

9.4 extract.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAXBUFFSIZE 1000000
#define BI_LEVEL 2
#define NLEN_BITS 16

/* static int buffer; *//* bits buffered for output */
extern unsigned char buffer;
extern int bits_to_go;/* Number of bits free/still in buffer */
extern int garbage_bits;/* Number of bits past end-of-file */

extern unsigned char *streamptr;
extern unsigned char *buf_start;
extern unsigned char *buf_end;

void start_outputing_bits();
void output_bit(int bit);
void done_outputing_bits();
void done_inputing_bits();
int input_bit();
void set_buffer (void *buffer);
void  *get_buffer ();
void start_inputing_bits();
unsigned int get_bits (int nbits);

int fileparse(unsigned char *inbuffer, int bitplaneOff,
              int resolutionOff, unsigned char *outbuffer);

int main(int argc, char *argv[])
{
  unsigned char *teststring;

  int i, bitplaneOff, resolutionOff;
  unsigned char *inbuffer, *outbuffer;
  FILE *infile;

  if (argc<4) {
    fprintf(stderr, "Usage: adapt InputFile bitplaneOff resolutionOff\n");
```

```
    exit(-1);
  }

  if ((infile=fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open input file\n");
    exit(-1);
  }

  bitplaneOff = atoi(argv[2]);
  resolutionOff = atoi(argv[3]);

  if ((inbuffer = (unsigned char*) calloc(MAXBUFFSIZE, sizeof(char))) == NULL)
  {
    fprintf(stderr, "Cannot allocate inbuffer, out of memory\n");
    exit(-1);
  }

  if ((outbuffer = (unsigned char*) calloc(MAXBUFFSIZE, sizeof(char))) ==
NULL)
  {
    fprintf(stderr, "Cannot allocate outbuffer, out of memory\n");
    exit(-1);
  }

  i = fread(inbuffer, sizeof(char), MAXBUFFSIZE, infile);

  i = fileparse(inbuffer, bitplaneOff, resolutionOff, outbuffer);

  if (fwrite(outbuffer, sizeof(char), i, stdout) != i) {
    fprintf(stdout, "Writing tempBuffer file: failed\n");
    exit(-1);
  }

  return 1;

}


int fileparse(unsigned char *inbuffer, int bitplaneOff,
              int resolutionOff, unsigned char *outbuffer)
{
  unsigned char *inptr, *outptr;

  int wvtDecompLev;
  int Color;
  int Ymean, Umean, Vmean;
  int QdcY, QdcUV;
  int DCsizeY, DCsizeU, DCsizeV;
  unsigned char *currbuffer;
  int i,j,l;

  int spatial_scalability_levels;
  int quantization_type;
  int SNR_length_enable;

  int SPlayer_Y_spatial_bitstream_length;
  int SPlayer_U_spatial_bitstream_length;
  int SPlayer_V_spatial_bitstream_length;
  int SPlayer_Y_SNRlayer_Quant;
  int SPlayer_U_SNRlayer_Quant;
  int SPlayer_V_SNRlayer_Quant;

  int SPlayer_Y_SNRlayer_snr_bitstream_length;
  int SPlayer_U_SNRlayer_snr_bitstream_length;
  int SPlayer_V_SNRlayer_snr_bitstream_length;
  int SPlayer_Y_SNR_scalability_levels;
  int SPlayer_U_SNR_scalability_levels;
```

```
int SPlayer_V_SNR_scalability_levels;

int len;

int bytes;
int bsize;
long temp;
long bitstream_size;
int Base;

//  Base = (int) pow(2., NLEN_BITS-1);
Base = 32768;

/* set the buffer */
set_buffer (inbuffer);
start_inputing_bits();

inptr = inbuffer;
outptr = outbuffer;

/* wavelet decomposition level (8 bits)
   output the level for luminance, the chrominance level is 1 less */
wvtDecompLev = get_bits(8);
*outptr++ = (unsigned char) *inptr++;

/* decide if the image is color or gray scale (8 bits) */
Color = get_bits(8);
*outptr++ = (unsigned char) *inptr++;

/* obtain Ymean, Umean, and V_mean (8 bits) */
Ymean = get_bits(8);
*outptr++ = (unsigned char) *inptr++;

if (Color) {
  Umean = get_bits(8);
  *outptr++ = (unsigned char) *inptr++;
  Vmean = get_bits(8);
  *outptr++ = (unsigned char) *inptr++;
}

/* obtain quantization step for DC band (Lum, Chrominance) (8 bits) */
QdcY = get_bits(8);
*outptr++ = (unsigned char) *inptr++;

if (Color) {
  QdcUV = get_bits(8);
  *outptr++ = (unsigned char) *inptr++;
}

/* get the length of the DC bitstream Y (16 bits) */
DCsizeY = get_bits(16);
*outptr++ = (unsigned char) *inptr++;
*outptr++ = (unsigned char) *inptr++;

/* Get the DC bitstream */
currbuffer = (unsigned char *) get_buffer();
for (i=0;i<DCsizeY/8;i++) {
  *outptr++ = (unsigned char) *inptr++;
  currbuffer++;
}
set_buffer (currbuffer);

if (Color) {
  /* get the length of the DC bitstream U (16 bits) */
  DCsizeU = get_bits(16);
  *outptr++ = (unsigned char) *inptr++;
  *outptr++ = (unsigned char) *inptr++;
```

```
    /* Get the DC bitstream */
    currbuffer = (unsigned char *) get_buffer();
    for (i=0;i<DCsizeU/8;i++){
      *outptr++ = (unsigned char) *inptr++;
      currbuffer++;
    }
    set_buffer (currbuffer);

    /* get the length of the DC bitstream V (16 bits) */
    DCsizeV = get_bits(16);
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;

    /* Get the DC bitstream */
    currbuffer = (unsigned char *) get_buffer();
    for (i=0;i<DCsizeV/8;i++){
      *outptr++ = (unsigned char) *inptr++;
      currbuffer++;
    }
    set_buffer (currbuffer);
  }

  spatial_scalability_levels = get_bits(5);
  quantization_type = get_bits(2);
  SNR_length_enable = get_bits(1);
  *outptr++ = (unsigned char) *inptr++;

  if (quantization_type == BI_LEVEL)          /* bi-level quantizer */
  {
    if (spatial_scalability_levels == 1){
      /* get the length */
      SPlayer_Y_spatial_bitstream_length = get_bits(24);
      *outptr++ = (unsigned char) *inptr++;
      *outptr++ = (unsigned char) *inptr++;
      *outptr++ = (unsigned char) *inptr++;

      /* get Quant */
      SPlayer_Y_SNRlayer_Quant = get_bits(16);
      *outptr++ = (unsigned char) *inptr++;
      *outptr++ = (unsigned char) *inptr++;

      /* get the bitstream for Y */
      currbuffer = (unsigned char *) get_buffer();
      for (i=0;i<SPlayer_Y_spatial_bitstream_length/8;i++){
       *outptr++ = (unsigned char) *inptr++;
       currbuffer++;
      }
      set_buffer (currbuffer);

      if (Color) {
       /* U component */
       /* get the length */
       SPlayer_U_spatial_bitstream_length = get_bits(24);
       *outptr++ = (unsigned char) *inptr++;
       *outptr++ = (unsigned char) *inptr++;
       *outptr++ = (unsigned char) *inptr++;

       /* get Quant */
       SPlayer_U_SNRlayer_Quant = get_bits(16);
       *outptr++ = (unsigned char) *inptr++;
       *outptr++ = (unsigned char) *inptr++;

       /* get the bitstream for U */
       currbuffer = (unsigned char *) get_buffer();
       for (i=0;i<SPlayer_U_spatial_bitstream_length/8;i++){
         *outptr++ = (unsigned char) *inptr++;
```

```
      currbuffer++;
    }
    set_buffer (currbuffer);

    /* V component */
    /* get the length */
    SPlayer_V_spatial_bitstream_length = get_bits(24);
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;

    /* get Quant */
    SPlayer_V_SNRlayer_Quant = get_bits(16);
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;

    /* get the bitstream for V */
    currbuffer = (unsigned char *) get_buffer();
    for (i=0;i<SPlayer_V_spatial_bitstream_length/8;i++){
      *outptr++ = (unsigned char) *inptr++;
      currbuffer++;
    }
    set_buffer (currbuffer);
   }
}
else if (spatial_scalability_levels > 1) {
  /* Y component */
  /*      for (l=wvtDecompLev-1; l>=0; l--) { */
  for (l=wvtDecompLev-1; l>=0; l--) {
   /* get length */
   SPlayer_Y_spatial_bitstream_length = get_bits(24);
   if (l>=resolutionOff) {
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
   }
   else inptr += 3;

   /* get Quant */
   SPlayer_Y_SNRlayer_Quant = get_bits(16);
   if (l>=resolutionOff) {
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
   }
   else inptr += 2;

   if (SNR_length_enable == 0) {

     /* Y bitstream */
     currbuffer = (unsigned char *) get_buffer();
     for (i=0; i<SPlayer_Y_spatial_bitstream_length/8; i++) {
       if (l>=resolutionOff) {
         *outptr++ = (unsigned char) *inptr++;
       }
       else inptr++;
       currbuffer++;
     }

     /* set the buffer to the new position */
     set_buffer(currbuffer);
   }
   else {
     len = 0;
     /*     j=0; */
     /*     while (len < SPlayer_Y_spatial_bitstream_length) {*/
     for(j=0; j<wvtDecompLev+2; j++) {
       /* get SNR length */
```

```
      bsize = get_bits(NLEN_BITS);
      if ((l>=resolutionOff) && (j<wvtDecompLev+2-bitplaneOff)) {
        *outptr++ = (unsigned char) *inptr++;
        *outptr++ = (unsigned char) *inptr++;
      }
      else inptr += 2;
      bitstream_size = (bsize>=Base)?bsize-Base: bsize;
      i=0;
      while (bsize >= Base) {
        i++;
        bsize = get_bits(NLEN_BITS);
        if ((l>=resolutionOff) && (j<wvtDecompLev+2-bitplaneOff)) {
         *outptr++ = (unsigned char) *inptr++;
         *outptr++ = (unsigned char) *inptr++;
        }
        else inptr += 2;
        temp = (bsize>=Base)?bsize-Base: bsize;
        bitstream_size = (temp<<(i*(NLEN_BITS-1)))+bitstream_size;
      }

      SPlayer_Y_SNRlayer_snr_bitstream_length = bitstream_size;
      len += SPlayer_Y_SNRlayer_snr_bitstream_length;

      /* get the bitstream */
      currbuffer = (unsigned char *) get_buffer();
      for (i=0; i<SPlayer_Y_SNRlayer_snr_bitstream_length/8; i++) {
        if ((l>=resolutionOff) && (j<wvtDecompLev+2-bitplaneOff)) {
         *outptr++ = (unsigned char) *inptr++;
        }
        else inptr++;
        currbuffer++;
      }
      /* set the buffer to the new position */
      set_buffer (currbuffer);

      /* j++;*/
    }  /* while () */
    /* SPlayer_Y_SNR_scalability_levels = j; */

 }  /* if (snr_len_enable) */
}

if (Color) {
 /* U component */
 /*     for (l=wvtDecompLev-2; l>=0; l--) { */
 for (l=wvtDecompLev-2; l>=0; l--) {
   /*     start_inputing_bits();*/
   /* get length */
   SPlayer_U_spatial_bitstream_length = get_bits(24);
   if (l>=resolutionOff) {
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
   }
   else inptr += 3;

   /* get Quant */
   SPlayer_U_SNRlayer_Quant = get_bits(16);
   if (l>=resolutionOff) {
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
   }
   else inptr += 2;

   if (SNR_length_enable == 0) {
     /* U bitstream */
     currbuffer = (unsigned char *) get_buffer();
```

```
    for (i=0; i<SPlayer_U_spatial_bitstream_length/8; i++) {
      if (l>=resolutionOff) {
       *outptr++ = (unsigned char) *inptr++;
      }
      else inptr++;
      currbuffer++;
    }
    /* set the buffer to the new position */
    set_buffer (currbuffer);
  }
  else {
    len = 0;
    /* j=0; */
    /* while (len < SPlayer_U_spatial_bitstream_length) { */
    for(j=0; j<wvtDecompLev+1; j++) {
      /* get SNR length */
      bsize = get_bits(NLEN_BITS);
      if ((l>=resolutionOff) && (j<wvtDecompLev+1-bitplaneOff)) {
       *outptr++ = (unsigned char) *inptr++;
       *outptr++ = (unsigned char) *inptr++;
      }
      else inptr += 2;
      bitstream_size = (bsize>=Base)?bsize-Base: bsize;
      i=0;
      while (bsize >= Base){
       i++;
       bsize = get_bits(NLEN_BITS);
       if ((l>=resolutionOff) && (j<wvtDecompLev+1-bitplaneOff)) {
         *outptr++ = (unsigned char) *inptr++;
         *outptr++ = (unsigned char) *inptr++;
       }
       else inptr += 2;
       temp = (bsize>=Base)?bsize-Base: bsize;
       bitstream_size = (temp<<(i*(NLEN_BITS-1)))+bitstream_size;
      }

      SPlayer_U_SNRlayer_snr_bitstream_length = bitstream_size;
      len += SPlayer_U_SNRlayer_snr_bitstream_length;

      /* get the bitstream */
      currbuffer = (unsigned char *) get_buffer();
      for (i=0; i<SPlayer_U_SNRlayer_snr_bitstream_length/8; i++) {
       if ((l>=resolutionOff) && (j<wvtDecompLev+1-bitplaneOff)) {
         *outptr++ = (unsigned char) *inptr++;
       }
       else inptr++;
       currbuffer++;
      }
      /* set the buffer to the new position */
      set_buffer (currbuffer);

      /* j++; */
    }  /* while () */
    /* SPlayer_U_SNR_scalability_levels = j; */

  }  /* if (snr_len_enable) */
}

/* V component */
/*     for (l=wvtDecompLev-2; l>=0; l--) { */
for (l=wvtDecompLev-2; l>=0; l--) {
  /* get length */
  SPlayer_V_spatial_bitstream_length = get_bits(24);
  if (l>=resolutionOff) {
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;
    *outptr++ = (unsigned char) *inptr++;
```

```
}
else inptr += 3;

/* get Quant Quant */
SPlayer_V_SNRlayer_Quant = get_bits(16);
if (l>=resolutionOff) {
  *outptr++ = (unsigned char) *inptr++;
  *outptr++ = (unsigned char) *inptr++;
}
else inptr += 2;

if (SNR_length_enable == 0) {
  /* Y bitstream */
  currbuffer = (unsigned char *) get_buffer();
  for (i=0; i<SPlayer_V_spatial_bitstream_length/8; i++) {
    if (l>=resolutionOff) {
     *outptr++ = (unsigned char) *inptr++;
    }
    else inptr++;
    currbuffer++;
  }
  /* set the buffer to the new position */
  set_buffer (currbuffer);
}
else {
  len = 0;
  /* j=0; */
  /* while (len < SPlayer_V_spatial_bitstream_length) { */
  for(j=0; j<wvtDecompLev; j++) {
    /* get SNR length */
    bsize = get_bits(NLEN_BITS);
    if ((l>=resolutionOff) && (j<wvtDecompLev-bitplaneOff)) {
     *outptr++ = (unsigned char) *inptr++;
     *outptr++ = (unsigned char) *inptr++;
    }
    else inptr += 2;
    bitstream_size = (bsize>=Base)?bsize-Base: bsize;
    i=0;
    while (bsize >= Base){
     i++;
     bsize = get_bits(NLEN_BITS);
     if ((l>=resolutionOff) && (j<wvtDecompLev-bitplaneOff)) {
       *outptr++ = (unsigned char) *inptr++;
       *outptr++ = (unsigned char) *inptr++;
     }
     else inptr += 2;
     temp = (bsize>=Base)?bsize-Base: bsize;
     bitstream_size = (temp<<(i*(NLEN_BITS-1)))+bitstream_size;
    }

    SPlayer_V_SNRlayer_snr_bitstream_length = bitstream_size;
    len += SPlayer_V_SNRlayer_snr_bitstream_length;

    /* get the bitstream */
    currbuffer = (unsigned char *) get_buffer();
    for (i=0; i<SPlayer_V_SNRlayer_snr_bitstream_length/8; i++) {
     if ((l>=resolutionOff) && (j<wvtDecompLev-bitplaneOff)) {
       *outptr++ = (unsigned char) *inptr++;
     }
     else inptr++;
     currbuffer++;
    }
    /* set the buffer to the new position */
    set_buffer (currbuffer);

    /* j++; */
  }  /* while () */
```

```
          /* SPlayer_V_SNR_scalability_levels = j; */
        }  /* if (snr_len_enable) */
      }
     }
   }
  }

  done_inputing_bits();

  /* how many bytes totally? */
  bytes = inptr - inbuffer;

  bytes = outptr - outbuffer;

  return (bytes);

}
```

9.5 akiyo.html

```
<html>

<head>
<title>
Wavelet Codec Image Demo -- akiyo image
</title>
</head>

<body>

<script LANGUAGE="JavaScript">
<!--
// Hide this script from old browsers --

var largeExpDate = new Date ();
var myDomain;

function getCookieVal (offset) {
   var endstr = document.cookie.indexOf (";", offset);
   if (endstr == -1)
      endstr = document.cookie.length;
   return unescape(document.cookie.substring(offset, endstr));
}

function GetCookie (name) {
        var arg = name + "=";
        var alen = arg.length;
        var clen = document.cookie.length;
        var i = 0;
        while (i < clen) {
                var j = i + alen;
                if (document.cookie.substring(i, j) == arg)
                        return getCookieVal (j);
                i = document.cookie.indexOf(" ", i) + 1;
                        if (i == 0)
                                break;
                }
   return null;
}

function SetParams()
{
  var bitplaneOff = GetCookie('bitplaneOff');
  var resolutionOff = GetCookie('resolutionOff');
```

```
  if ((bitplaneOff == null) || (resolutionOff == null))
  {
    bitplaneOff=0;
    resolutionOff=0;
  }

  document.write('<center>');
  document.write('<embed src="http://www.yourwebserver.com/nph-
adapt.pl?'+bitplaneOff+','+resolutionOff+'" width=352 height=288 ');
  document.write('bitplaneOff='+bitplaneOff);
  document.write(' resolutionOff='+resolutionOff);
  document.write(' type="application/x-waveletimage">');
  document.write('</center>');

}
SetParams();
// -- End Hiding Here -->
</script>

</body>

</html>
```

4980-46