JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©JOT, 2005

Vol. 4, No. 4, May-June 2005

Synthetic Image Sequence Compression

Douglas Lyon, Fairfield University, Fairfield CT, U.S.A.

Abstract

This paper describes a technique for compressing computer screen shots into a GIF animation file. The goal is to distribute the animations, to a variety of browsers, without requiring a plug-in or helper application. We seek to minimize the size of the image sequence, while maximizing the signal to noise ratio of the sequence. The GIF animation format has several constraints; images may have a maximum of 256 colors, and the images must all be of the same size. Further, to minimize overhead, we seek to make use of a single color lookup table for the entire animation.

Several color quantization algorithms are compared, using SNR (Signal to Noise Ratio) as the metric of quality (as well as subjective appearance).

We present the design of an interface, written in Java, and distributed freely using Java Web Start, that employs a well know neural network program and a color quantization algorithm to capture screen shots and save them to the GIF animation.

GIF animations represent silent movies, as they have no sound to accompany them. However, they are still in wide use and have applications in entertainment and education.

The techniques described are a part of the *JSnap* project, a joint project between the skunk works of DocJava, Inc. and Fairfield University.

1 INTRODUCTION

Synthetic images sourced from the screen snapshots are different from the typical image sequences sourced from standard sensors (video cameras, scanners, etc.). The images are characterized by having little to no sensor noise. Further, the images can typically range in size from 64x64 pixels to 1024x768 (or larger). Computer displays with 24-bit color depth are presently standard. Image sequences are typically displayed with a refresh rate of 60 Hz or better. However, for the purpose of distribution on the Web, only difference images need to be transmitted and the rate of image change can be up to several seconds per frame, depending on the material and the application.

One of the basic problems with Java (before JDK 1.3) is that there was no way to portably capture the screen without resorting to native method invocations. As of JDK

Cite this column as follows: Douglas Lyon: “Synthetic Image Sequence Compression”, in *Journal of Object Technology*, vol. 4, no. 4, May-June 2005, pp.19-31
http://www.jot.fm/issues/issue_2005_05/column02

1.3, a new class was introduced into the Abstract Windowing Toolkit (AWT) called the *Robot* class. The Robot class was designed for testing of GUI's and event processing. However, we have made use of it to perform screen captures in order to generate image sequences. Thus, the technique presented in this paper provides the enabling technology for others to acquire image sequence data and perform compression experiments.

GIF images are constrained to 256 colors (i.e., they are 8-bit images) [Murray]. Thus, we are faced with a sub-problem of converting 24-bit color images into 8-bit color images. This is called color requantization and requires that some colors be discarded from the input image and remapped into new color in the output image. There are many algorithms for performing color-requantization, and many criteria for determining the optimality of the algorithms.

1.2 Distortion Metrics

This section describes some common distortion metrics used to measure one aspect of a quantization algorithms performance. The mean-square distortion is, perhaps, one of the most common metrics. Suppose, for example, an input, x is quantized by a function called a quantizer, Q. The mean-square distortion is computed by taking the expectation of the square of the difference between the algorithms output value and the input value of a pixel, then multiplying by the probability of the value. In the continuous one-dimensional domain, we write

$$D = \int_{-\infty}^{\infty} \left[Q(x) - x\right]^2 p(x)dx \quad (1.1)$$

where

$$D = \text{mean-square distortion measure}$$
$$p(x) = \text{probability of value } x \text{ and}$$
$$Q(x) = \text{quantized value for } x$$

Typically, the quantizer's performance is measured using the signal-to-noise ratio (SNR), which is given in dB as

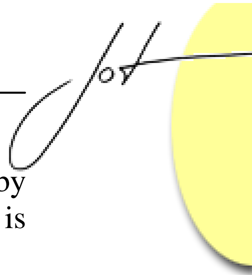
$$SNR_{dB} = 10\log_{10}(\sigma^2 / D) \quad (1.2)$$

and

$$\sigma^2 = \text{variance of the input}$$
$$= E(x^2) - \left[E(x)\right]^2$$

Where $E(x)$ is the expected value for x.

Unfortunately, distortion measures, such as the SNR, are not necessarily reflective of any physiological metric for improving the subjective appearance of an image. Hence their use is open to question. For example, histogram equalization has been shown to improve an image's appearance; however, according to (1.2) such a process will lower the SNR. The reason that the appearance is improved may actually have to do with the improved contrast ratio of the image. Such a subjective improvement is not taken into account with (1.2).



Given a discrete point set, we can modify (1.2) to reflect the distortion function by summing the Euclidean distances between the color of each pixel and its map. This is expressed in

$$D_{tqe} = \sum_{y=0}^{height-1} \sum_{x=0}^{width-1} e_{x,y}^2 \tag{1.3}$$

where

$$e_{x,y} = Q(C_{x,y}) - C_{x,y}$$
$$D_{tqe} = \text{ total quantization error}$$
$$C_{x,y} = \text{ color at location } x, y$$
$$Q(C_{x,y}) = \text{ quantized color at location } x, y$$

In fact, we could obtain the mean-square distortion measure from the total quantization error by dividing it by the total number of pixels, that is;

$$D = \frac{D_{tqe}}{width * height} \tag{1.4}$$

This is computed by subtracting the original image from the quantized image, squaring the resulting error pixels, summing their color components, then dividing by the total number of pixels. The mean square error (MSE) represented by (1.4) is a widely used measure of distortion and is also called the *coding noise power* [Netravali].

Another metric of coding performance is the bit rate. It is typical to seek to minimize the bit rate and the MSE. Typical of most engineering tradeoffs, the MSE is inversely related to the bit rate. Bit rate is function of the number of bits needed per pixel. As this can change from pixel to pixel, and image to image, one method for computing bit rate is to measure the image (or image sequence) file size, in bits, then divide by the total number of pixels. This takes into account overhead in writing out the file, in any given format.

Perceptual difficulties aside, it is useful to have an objective fidelity citerion, such as the SNR, to use when evaluating a lossy coding scheme. In addition, SNR is one of the most used fidelity criteria. One way to compute the SNR in dB is

$$SNR_{dB} = 10 \log_{10} \left[\frac{1}{D_{tqe}} \sum_{y=0}^{height-1} \sum_{x=0}^{width-1} Q(C_{x,y}) \right] \tag{1.5}$$

The SNR defined in (1.5) is consistent with [Myler] and can also be used on image compression algorithms (where the quantized image is replaced with the compressed image).

1.2. Color Quantization of Still Images

There are several techniques available for reducing the number of colors (i.e. dynamic range) in an image (or image sequence). A simple, fast method (still in wide use) is called

the *linear cut* algorithm. It works by cutting off bits from the pixels' least significant bits first, in the integral RGB color space.

Consider the integral RGB color space. Each component is constrained to range from 0 to 255 and resides in a 16 bit short array. To perform the linear cut algorithm on such an array, we need to mask the low-order bits that we want to "cut" out of the pixel, for example:

```
public void linearCut(short a[][], int numberOfBitsToCut)
  {
  int mask = 255 << numberOfBitsToCut;
  for (int x=0; x < width; x++)
    for (int y=0; y < height; y++)
          a[x][y] = (short)(a[x][y] & mask);
}
```

The mask in the *linearCut* method is computed, assuming that there are only 8-bits per color. If the programmer performs a linear cut of the last two bits, for example, we shift left 2 bits so that the least significant two bits are cleared (e.g., 11111111 << 2 becomes 1111111100). Thus the mask will thus remove the least significant two bits from the short stored in a[x][y].

Another algorithm in common use is called the *median cut* algorithm [Heckbert 82]. The goal of the median-cut algorithm is to have each color represent approximately the same number of pixels.

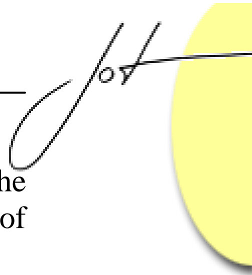
The algorithm is implemented by first computing a color histogram of the image in RGB color space. The histogram is then clustered into *k* groups. Once the clustering is performed, the pixels are mapped to the centroids of the clusters in order to minimize the color error in the image. A tightly fitting color cubed is created for the color space. It is then cut at the median of the longest axis (hence the name, *median cut* algorithm). The median cut procedure is applied to subcubes until there are *k* cubes. The centers of the cubes are used for the *k* output colors in the color map. Given a color map, each pixel is mapped from its original color to its nearest color neighbor. This mapping is done to minimize the color error metric given by (1.3).

One way to implement the median cut algorithm is to use a queue to enable a breadth-first cutting of the sub-cubes. The idea is that an instance of the *Cube* class has a static member variable that is used to keep track of the total number of *Cube* instances. The pseudo-Java follows:

```
int k = 256 // number of colors
CubeQueue cq = new CubeQueue();
//get a cube that fits tightly around the list of colors.
Cube c = new Cube(colorList);
cq.enqueue(c);

while (c.ncubes < k) {
   c = cq.dequeue();
   Cube childArray[] c.split();
   cq.enqueue(childArray[0]);
   cq.enqueue(childArray[1]);
}
```

The use of the *CubeQueue* is for illustration only. The implementation in the *MedianCut* class (as distributed by [Lyon 99]) does not make use of explicit recursion, due, in part, to the computational expense. In fact, given the a priori knowledge of the number of cubes



needed, we allocate an array that is exactly *k* cubes in length. Then we simulate the queue, using an array index. Once the list of sub-cubes is known, it is a matter of assigning the colors to the centroids of the cubes that minimize the *color error*.

As initially formulated by Heckbert [Heckbert 80], the color error is measured as the Euclidean distance from the pixel's original color to the remapped color. The sum of all the errors in the remapping is the objective function whose minima is sought. In fact, this is a standard metric in *clustering*.

In clustering, we are given a set of points in a Euclidean space and are asked to group them into *k* partitions to minimize a distortion function.

Two other algorithms we considered include the Wu color quantization algorithm and the Octree encoding algorithm. Java implementations for these algorithms are detailed in and available from http://www.docjava.com [Lyon 99] [Wu][Wu 97].

Converting to 256 colors, the test image, using Wu color quantization, yielded 45.1 dB SNR. Octree encoding yields a 42 dB SNR on the sample image. Linear cut yields a 32 dB SNR while Median cut yields a 45.2 dB SNR. The winner was the neural network color quantization algorithm, at 51 dB SNR.

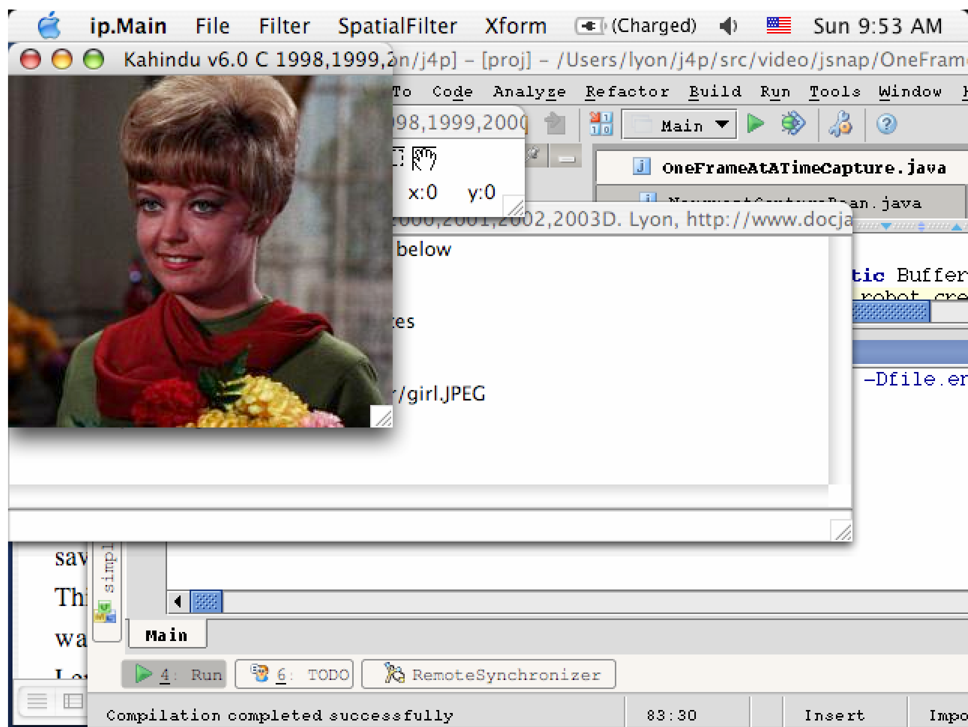


Figure 1. Original Picture

Figure 1 shows the original picture, with over 300k different colors. The image was taken using a 24-bit color display that included a standard color test image. The over all image is 640x480 pixels in size.

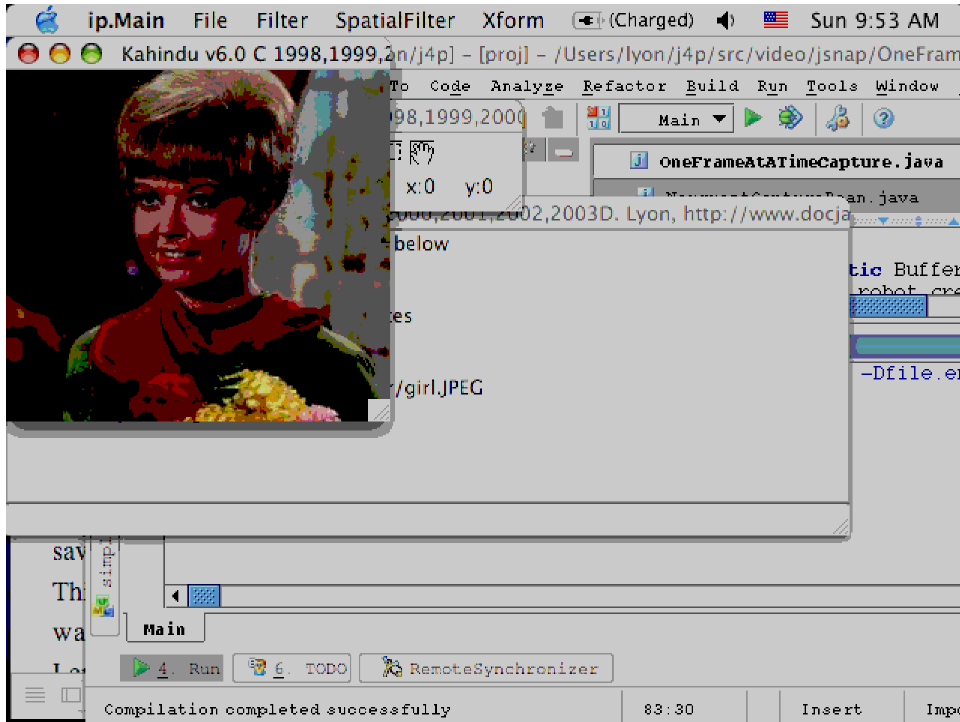


Figure 2. The linear cut algorithm

Figure 2 shows the original test image after the application of a linear cut algorithm. The image was reduced to 9 bits (3 bits per pixel).

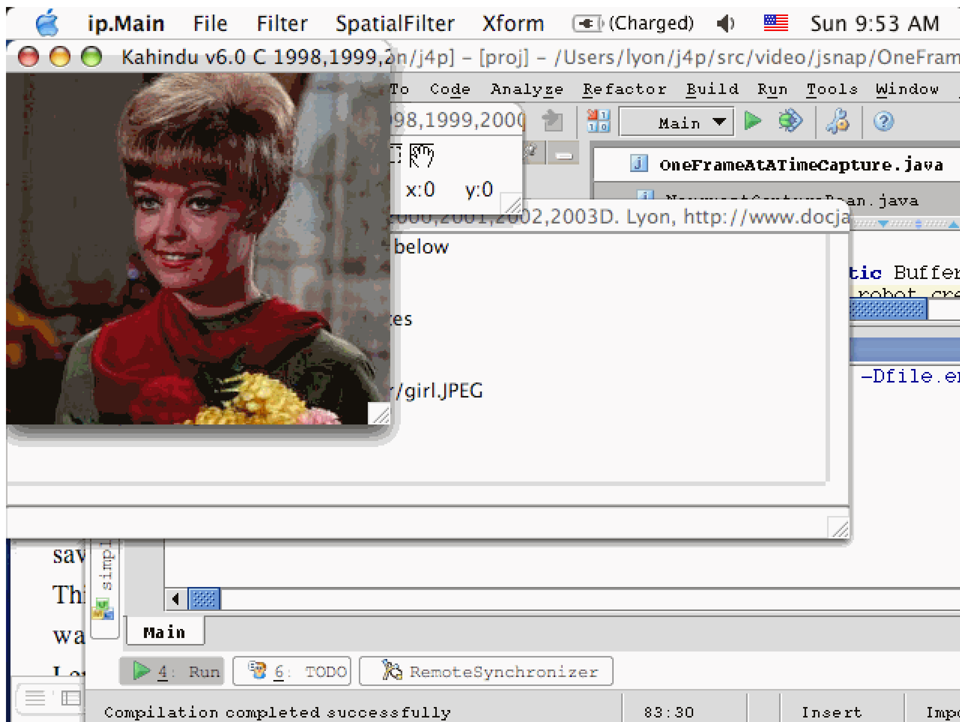
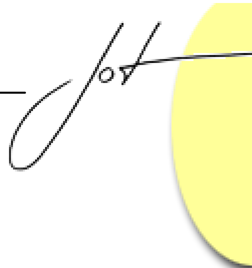


Figure 3 Octree color reduction

Figure 3 depicts the test image after being color quantized to 256 colors using the Octree algorithm.



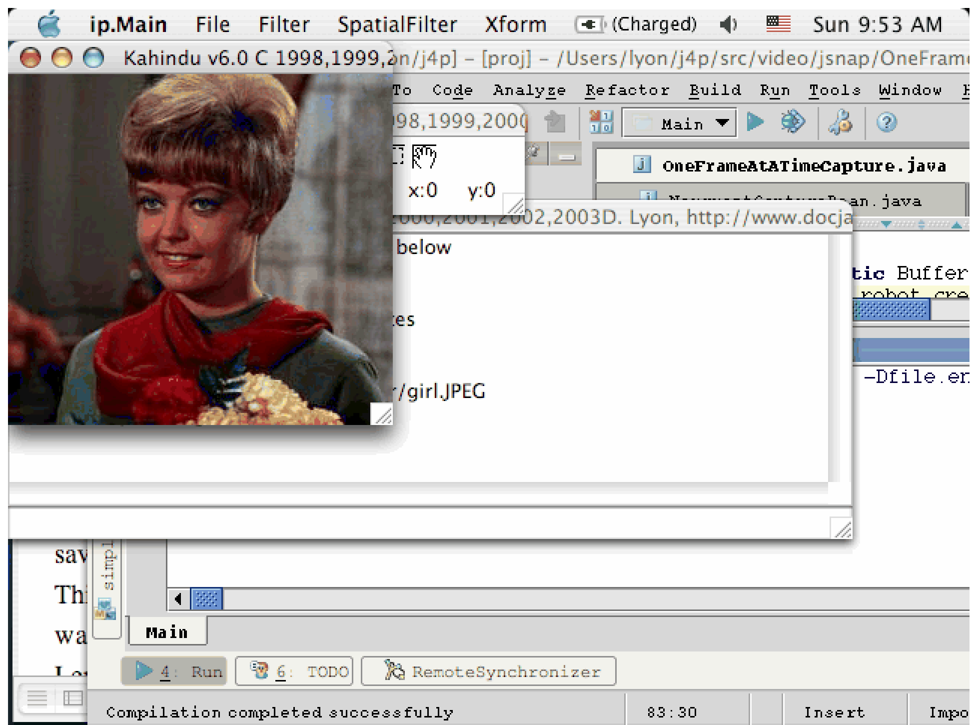


Figure 4. The Median Cut algorithm

Figure 4 shows the test image after application of Heckbert's median cut algorithm. The image was reduced to 256 colors.

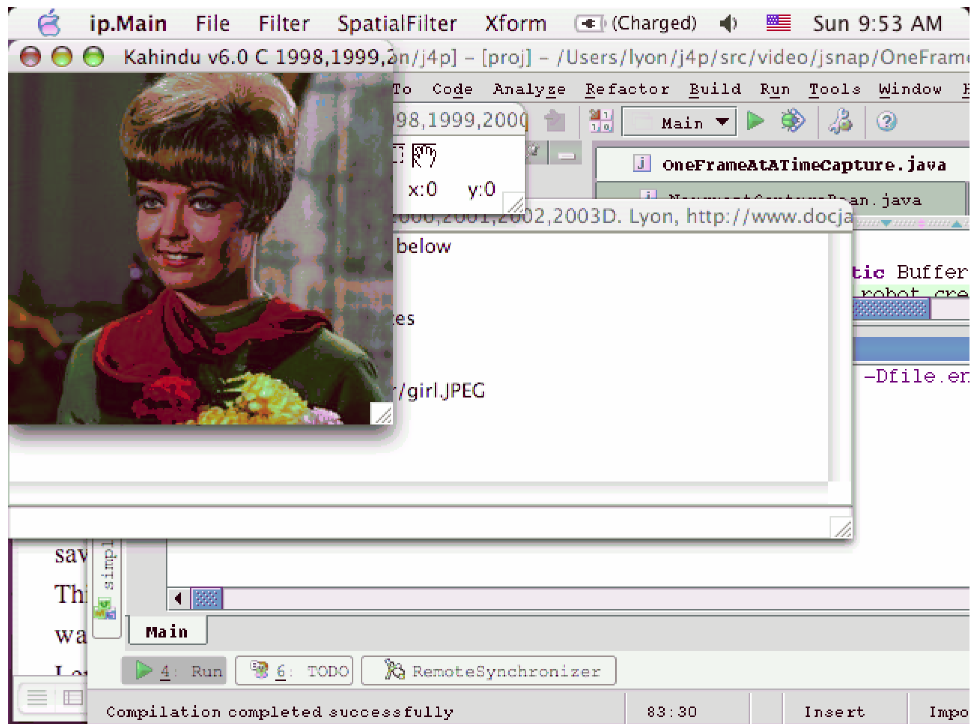


Figure 5. Wu color quantization algorithm

Figure 5 shows the original image after it has been quantized to 256 colors using Wu's color quantization algorithm.

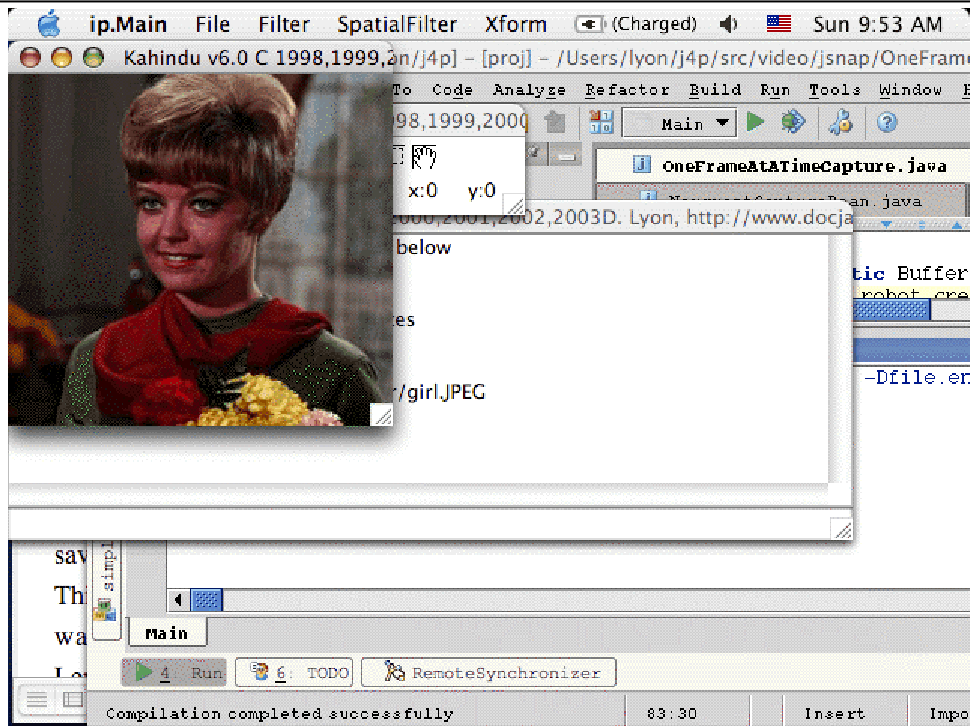


Figure 6. Neural Network quantization algorithm

Figure 6 shows the original image after it has been quantized to 256 colors using the neural network algorithm presented by [Dekker]. As a result of the improved SNR, we have selected the neural network algorithm for the image sequence compression.

1.3 The JSnap Application

In this section we describe the interface and operation of the JSnap application.

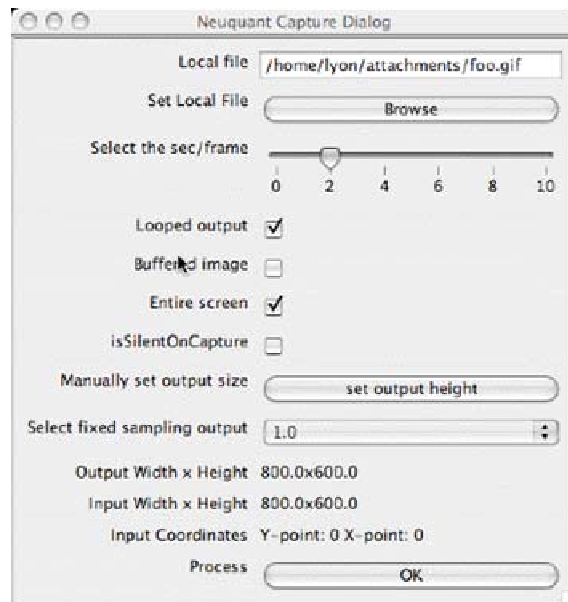
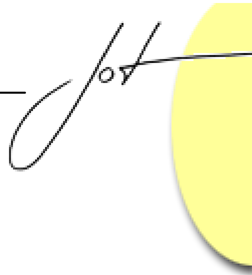


Figure 7. The JSnap Application Interface

The *JSnap* GUI provides a feature to enable fixed-size resampling of the screen using an output combo-box menu items, as shown in Figure 8.



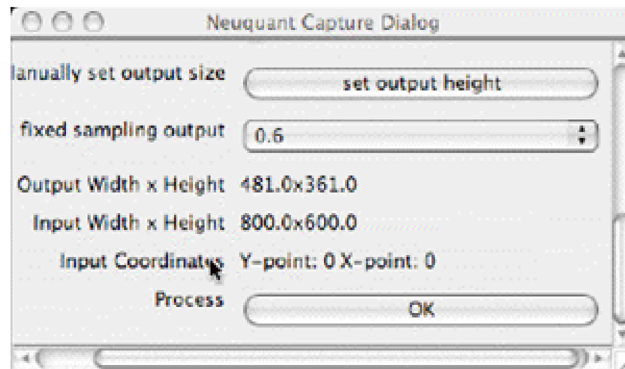


Figure 8. Setting the output side using a Combo-box

As an alternative you can manually set the output size by dragging a rectangle across the screen.

2 THE IMAGE SEQUENCE COMPRESSION ALGORITHM

The image sequence color requantization algorithm makes the assumption that the color look-up table for the first image is going to be used for all of the following images. The impact of such an assumption is that images will be degraded in their appearance if they contain colors that are radically different from the initial image. The algorithm is based on an API authored by Kevin Weiner [Weiner].

Given a color lookup table, a pixel is remapped to a color that is closest according to the minimum mean square error. This is performed using a search through all 256 colors in the color lookup table. After the color requantization, a difference frame is computed and the pixels are Lempel-Ziv compressed using the algorithm described in [LZ77].

The hybrid algorithm employed in this paper first color reduces the first image using the Wu color quantization algorithm [Wu 97]. The first image is then used to establish the color look-up table for subsequent images. To speed the color mapping to one of the selected colors in the look-up table, a neural network algorithm was employed [Dekker]. This is because the combination of the two algorithms appears to be better than either algorithm by itself. Also, the subjective appearance was used to make this judgment, not the SNR.

3 EXPERIMENTAL RESULTS

In this experiment we take a series of screen snapshots at 800x600 resolution, converting them to a GIF animation. We then measure the size of the file produced and compare this against the 24-bit estimate of what the size of the image sequence would be, without compression. That is, for the 10 images, we would need 800x600 pixel * 3 bytes per pixel * 10 images = 14.4 MB. The GIF animation produced is 490,336 bytes (29:1 compression ratio). Or, to put it another way, we had (490,336*8) bits / (800x600x10) pixels to obtain 0.8 bits per pixel, on average. As GIF images are internally Lemple-Ziv

compressed (after the difference frames are encoded), using utilities, like Gzip provides little benefit (490k is reduced to 487k). On the other hand, there are utilities that will recompress the *JSnap* output from 490k bytes to 392k bytes [ASG] promising no change in appearance. The question of why the program is out-performed by the utility design for this purpose remains open. It has been suggested that there is an additional advantage to be had in the software implementation by selecting for an appropriate "transparent" color in the GIF sequence, however, experiments in this direction have yet to validate this assertion. Thus, there is room for improvement.

One attribute, speed, was not addressed in the previous metrics of algorithmic performance. That is due, in part, to the stop-and-wait nature of the application. That is, the image sequence author must set up the next frame and then indicate to *JSnap* that the frame is ready for capture. Manual set-up time can range from a few 10's of seconds to several minutes, depending on how much work is required. JSnap is able to take a snapshot of the screen and add it to the GIF animation, at 800x600 pixels resolution, in between 0.5 and 3 seconds per frame (on a rather slow 800 Mhz G4 laptop). As of this writing, machines that are 2 and 3 times faster are commonly available, and so speed of execution seems acceptable, for this application.

Another metric of performance is the subjective appearance of the image, to the viewer. This is perhaps, the most important metric, yet it is also one of the hardest to quantify.

Figure 9 shows the original image of Figure 1, after being resampled using *JSnap* and requantized to look-up table of the original image sequence. Since the look up table was established for an image other than the original image, we observe significant degradation as a result of the output.

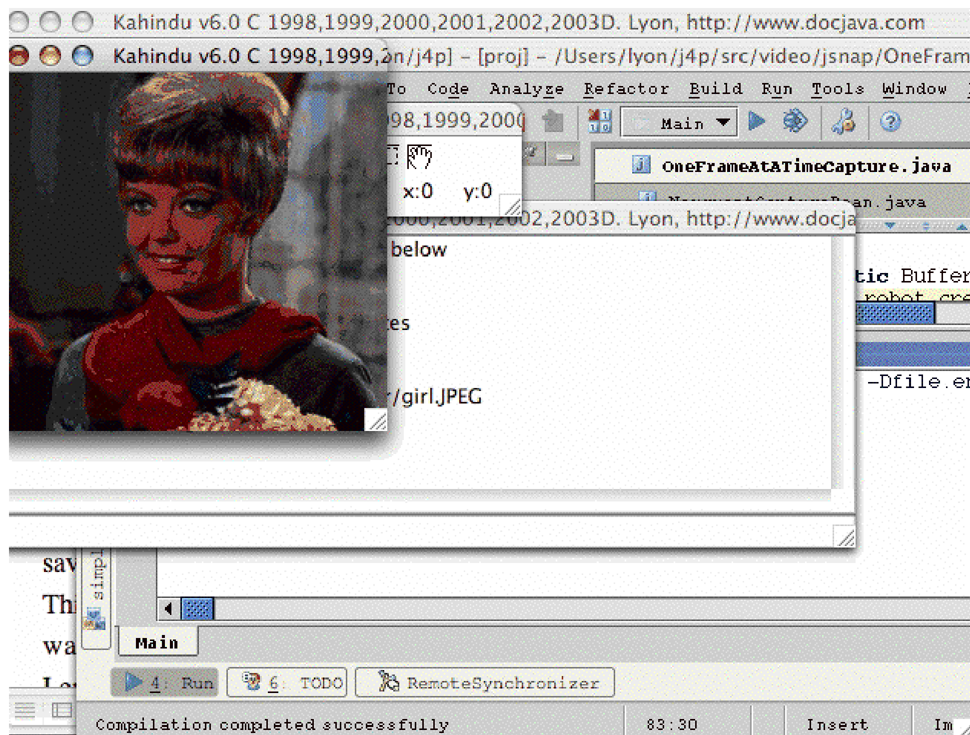
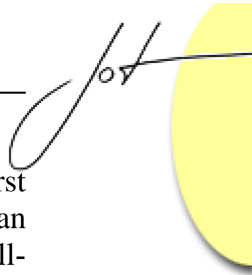


Figure 9. The Original Image After Requantization by JSnap



On the other hand, if the original image is used as the start image, the output for the first image is almost identical to the initial image, for a single frame, and the SNR for such an image is 33 dB. Thus the use of the algorithm can go from being nearly as good as a still-frame requantization to being far worse.

4 CONCLUSION

The algorithm used to perform image sequence color requantization makes use of the first image in the sequence. Since the first image will probably not represent the color composition of the following images, a more sophisticated algorithm might, in general give a better SNR, on average. An alternative, a more computationally expensive algorithm could insert extra color look up tables, if the SNR falls below a given threshold. In addition to the computational cost of such an approach, there is the cost of inserting a new color lookup table and of transmitting a frame that contains all the pixels (rather than a difference frame), as well as the cost of computing the SNR. This type of algorithm is left for future work.

Given a color lookup table, a pixel is remapped to a color providing a minimum mean square error. This is performed using a neural network search through the color lookup table. This is probably where the algorithm could benefit most from an algorithm that speeds the search for a best color. Aside from nearest neighbor algorithms based on the construction of a three dimensional Voronoi diagram, there are hash table algorithms for mapping colors into efficient color tables that could be exploited [Lyon 99]. As an alternative, the color look up table can be mapped into a binary tree key derived by requantization of luminance. Efficient inverse color map computation is addressed in [Thomas]. Eppstein also has a fast hierarchical clustering algorithm that also appears to hold promise [Eppstein]. The exploration of such algorithms is left as a topic of future work.

REFERNCES

[ASG] A Smaller GIF, available from http://www.peda.com/smaller /.

[Dekker] "Kohonen neural networks for optimal colour quantization", by Anthony Dekker, *Network: Computation in Neural Systems*, Vol. 5, 1994 pps. 351-367.

[Eppstein] "Fast Hierarchical Clustering and Other Applications of Dynamic Closest Pairs", *The ACM Journal of Experimental Algorithmics*, by David Eppstein, University of California at Irvine, vol. 5, No. 1., 2000, pps. 1-23. Available from http://www.jea.acm.org/2000/EppsteinDynamic/

[Heckbert 80] Color Image Quantization for Frame Buffer Display, by Paul S. Heckbert, B.S. Thesis, 1980, Architecture Machine Group, MIT, Cambridge, MA. Available at http://www.cs.cmu.edu/~ph .

[Heckbert 82] "Color Image Quantization for Frame Buffer Display", by Paul Heckbert, *Computer Graphics*, vol. 16, No. 3, July. 1982, pps. 297-307. Available at http://www.cs.cmu.edu/~ph .

[Lyon 99] Douglas A. Lyon, *Java for Programmers*, Prentice Hall, Upper Saddle River, NJ, 07458. 1999. Available from http://www.docjava.com .

[LZ77] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, vol. 23, No. 3, pp. 337-343.

[Murray] *Graphics File Formats*, by James D. Murray and William Vanryper. O'Reilly & Associates. CD, 1996.

[Myler] *Computer Imaging Recipes in C*, by Harley R. Myler and Arthur R. Weeks. Prentice Hall, Englewood Cliffs, NJ. 1993

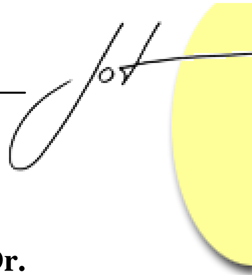
[Netravali] *Digital Pictures*, by Arun Netravali and Barry Haskell. Plenum Press, NY. 1988.

[Thomas] "Efficient Inverse Color Map Computation" in *Graphics Gems Volume II*, Adademic Press, Inc., Cambridge, MA, 1991, pps.116-125

[Weiner] *AnimatedGifEncoder* by Kevin Weiner, FM Software, kweiner@fmsware.com.

[Wu 97] "Lossless Compression of Continuous-Tone Images via Context Selection, Quantization and Modeling", by Wu. IEEE Transactions on Image Processing, vol. 6, No. 5. May. 1997, pps. 656-664.

[Wu] "Efficient Statistical Computations For Optimal Color Quantization", by Xiaolin Wu, in Graphics Gems, vol. II, edited, by James Arvo, Academic Press, Inc., Cambridge, MA. 1991, pp. 126-133.



About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.