# Assignment 7: Wavelet Compression

**CS 148 Autumn 2012-2013**

*Due Date: Wednesday, November 14th, 2012 at 7pm*

(a) Original image (b) Quantized image (c) Reconstructed image

Figure 1: Three steps of the image compression process

**Introduction** The goal of this assignment is to learn the uses and importance of wavelets through the implementation image compression and reconstruction using the Haar wavelet transform. In this assignment, you will perform a simple Haar wavelet transform in order to compress an image. An excellent introduction and description to the Haar transform can be found in Wavelets for Computer Graphics: A Primer Part 1. You should definitely look at Figure 5 and 6 on page 5, which describe the differences between the standard and nonstandard decomposition of an image. In addition to implementing both the forward and inverse Haar wavelet transformations and the quantization and reverse quantization steps, you'll need to compute the PSNR value and the difference image to compare the original image with the reconstructed image. From the previous assignments, you should know how to create a GUI application. You should also have all the libraries you need for this assignment. Thus, for this assignment, we will not be providing starter code.

## Background - Understanding the Algorithm

**Image Compression** Given an input image we want to transform it to make it more amenable to compression. There are 3 key steps to this process:

- Transformation: Transform the data onto a different basis/representation, so that it may be compressed more readily.
- Quantization: Reduce the bit-depth of the data. This is the main lossy step, and the result is an approximation of the original data.
- Coding: Store the data efficiently

Since the quantization step is the main lossy step, we're going to add a user controlled parameter to control the quantization, which, in turn, controls the quality and compressed data size. For this assignment we are not going to perform the coding. Instead, we'll calculate the entropy of the input and the quantized data, which is a close approximation to the encoded data size. In addition, we'll be creating a difference image and computing the PSNR of the reconstructed image. For the definition of PSNR, see below.

**Review of the Haar Wavelet Transform:** As discussed in lecture, wavelets are a tool for *multiresolution analysis*. The Haar wavelet transform is the simplest wavelet and straightforward to implement. Each step of the wavelet transform takes an input signal of length $n$ and generates a smoothed (lower resolution) version of length $n/2$ and a high frequency detail signal of length $n/2$. We can then continue to apply the same transform to the half size lower resolution signal to get an even lower resolution image and lower frequency details. The Haar transform in particular is simple because it generates these smoothed and detail signals in a simple manner. Given an input signal, each pair of terms `input[2*i]` and `input[2*i+1]` form a single smoothed value `smooth[i]` and a single detail value `detail[i]`, where `i` is in the range $[0, \text{length}/2)$. Given the input values $A$ and $B$, the smoothed and detail values we output are:

$$S = \frac{(A+B)}{2} \tag{1}$$

$$D = \frac{(A-B)}{2} \tag{2}$$

If $A$ and $B$ are both in the range $[0, 1]$, the formula for $S$ works well because the average value of $A$ and $B$ will always be in the range $[0, 1]$ and represent a valid subsampled image. On the other hand, the difference term is always in the range $[-.5, .5]$. To simplify quantization, we can just add .5 to the difference term to make sure that all values lie in the range $[0, 1]$:

$$S = \frac{(A+B)}{2} \tag{3}$$

$$D = \frac{(A-B)}{2} + .5. \tag{4}$$

The inverse transform, given $S$ and $D$ values, is given by

$$A = S + (D - .5) \tag{5}$$

$$B = S - (D - .5) \tag{6}$$

which can be easily verified by substitution.

The 2D version of the wavelet transform can be expressed as a series of 1D transforms on the rows and columns of the image. We'll use the *non-standard decomposition* in this assignment, which is expressed, as pseudo-code, by the following:

```
for wavelet level = log_2(image_size) to 1 {
    for each row i {
        wavelet_transform_step(row(i));
    }
    for each column j {
        wavelet_transform_step(column(j));
    }
}
```

The inverse 2D transform is similar, but works backwards from the lowest to highest level and performs inverse wavelet transform steps.

**Questions** In addition to reading the paper Wavelets for Computer Graphics: A Primer Part 1, we recommend you answer the assignment questions before you start writing the code.

1. Consider a 2-by-2 pixel image with pixels $A$, $B$, $C$, and $D$ as shown in the following diagram.

| | |
|---|---|
| $A$ | $B$ |
| $C$ | $D$ |

   What is the result of applying one iteration of the forward 2D transform? Your answer should be a 2-by-2 matrix.
2. In Section 2.2 of the Wavelets for Computer Graphics: A Primer Part 1, how is the vector space $V^j$ defined? How does it represent an image of $2j$ pixels?
3. Give an intuitive argument using one dimensional images that the vector space $V^j$ is contained in vector space $V^{j+1}$. (What is a vector in the $V^j$ space? What is a vector in $V^{j+1}$?)
4. Compare nonstandard image decomposition with the standard one. Why are we using the nonstandard image decomposition instead of the standard one?

## Steps

There is a lot to do in this assignment, so we'll do our best to break it down into a nice, simple checklist for you here.

**Step 1: Download Test Images** Download the test images: `Mandrill.png`, `MonaLisa.png`, `Tahoe.png`. Note that you're required to test your program on different images and show screenshots during your demo. *In this assignment, you will need to create your own wavelet compression application. No starer code is provided.*

**Step 2: Read Assignment Specification** Your application should successfully perform the following tasks.

1. Load `MonaLisa.png` by default. It should also be able to take a command-line argument (i.e. `./Wavelet imageFilename`) to change the image to load. (This makes it easier for us to compare your wavelet compression results.)
2. Reject any images that is not of size $2j$ for some positive $j$. Note that the Haar transforms are only defined for images with power of 2 width and height.
3. Allow for four image objects: the original image, the transformed-quantized image, the reconstructed image, and the difference image.
4. Display four images: the original image, the transformed-quantized image, the reconstructed image, and the difference image.
5. Perform the 2D Haar wavelet forward and inverse transforms using the non-standard decomposition.
6. Perform quantization to convert transformed floating point data to compressed stream (`unsigned char`).
7. Perform reverse quantization on the compressed stream to get back an approximation of the floating point data.
8. Allow the user to control the quality parameter for your compression by keyboard. One option is to have the user increase and decrease quality using the `a` and `d` keys respectively. Update the displayed images according to the value of the quality parameter.
9. Print the quality parameter, PSNR, and the entropy terms to the console when you adjust the compression quality parameter.

A screenshot of what the application should look like is provided in Fig. 2. If you do not know where to start, here are some more hints:

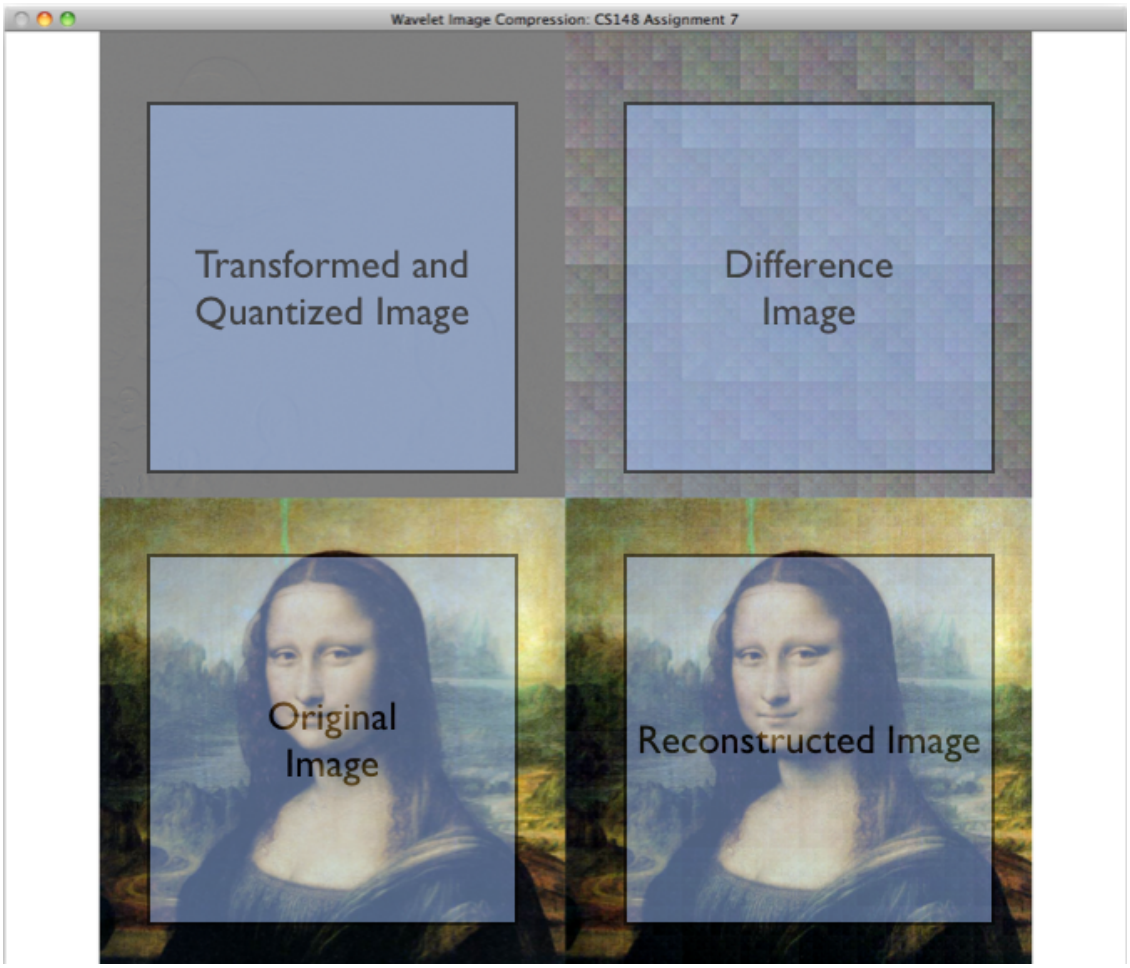


Figure 2: Screenshot of final application. Note that the blue text boxes shown above are just for illustration. You only need to display the 4 images. *Do not draw text boxes* – We want to see your images clearly.

1. Create a directory for your wavelet application. If you are using `libST`, please have `libST` compiled in a sibling directory to your `Wavelet` directory, similar to the previous assignments. See the Hints section for setting up `libST` with Visual Studio or Xcode.
2. Your should implement `main.cpp` to handle your GLUT interaction.
3. Implement a `Wavelet` class so that your application can load the images when the application launches and display four images within the application window: the original image, the transformed-quantized image, the reconstructed image, and the difference image. These 4 images can be initialized to the original image.

We also provide you examples of the function interface for wavelet transforms, but you do not need to use them as long as you satisfy the assignment requirement.

```
void HaarTransformForward1D(const STVector3* signal, STVector3* transformed);
void HaarTransformInverse1D(const STVector3* transformed, STVector3* signal);
void HaarTransformForward2D(const STVector3* signal, STVector3* &Transformed);
void HaarTransformInverse2D(const STVector3* transformed, STVector3* &signal);
void Quantize(const STVector3* exactValues, STImage* quantizedValues);
void Unquantize(const STImage* quantizedValues, STVector3* approximateValues);
float ComputeEntropy(const STImage* image);
float ComputePSNR();
void ComputeDifferenceImage();
```

**Step 3: Implement Haar Wavelet Transform: Forward and Inverse** It is easier to start from the one-dimensional Haar wavelet transform.

1. Haar Wavelet Transform: 1D
   - Write code to perform a 1D forward Haar wavelet transform. In each 1D transform step, it should compute the averages and differences and store them (averages in the first half, differences in the second) in the output array.
   - Write code to perform a 1D inverse Haar wavelet transform. This should be a functional inverse of the 1D forward Haar transform.
2. Haar Wavelet Transform: 2D
   - Write code to perform a 2D forward Haar wavelet transform using the **non-standard decomposition**. The 2D Haar wavelet transform should perform a series of 1D Haar transform steps.
   - Write code to perform a 2D inverse Haar wavelet transform. This should be a functional inverse of the 2D forward Haar transform.

If you apply the `HaarTransformForward2D(...)` and the `HaarTransformInverse2D(...)` on an image in sequence, you should get the same image as the original one. The forward and inverse functions are so similar that you may want to copy the code from the forward to the inverse and then make the necessary changes

**Step 4: Implement Quantization and Reverse Quantization** Your program should let the user have control over the quality of the compressed image.

1. **Quantization** Given an `STVector3` that we calculated in the forward wavelet transform steps we need to quantize it to store it efficiently. Because of the way we constructed the wavelet transform, all the detail terms are in the range $[0, 1]$. Because we've guaranteed our wavelet transformed data is all in the range $[0, 1]$ the easiest way to quantize is to simply convert back to a `Pixel`. `Pixel`s use `unsigned char`s for storage, so we are quantizing to 8 bits. However, we want some control over the quality. As discussed in class, we can devote fewer bits to higher subbands (higher frequencies) by dividing those terms by larger values so that they take on a smaller range of values. One example of a quantization factor you can use is:

```
output_i = (p_i - 0.5) / (level^(qfactor^3)) + 0.5
```

where `output_i` is the output `STColor`, `p_i` is the input `STVector3`, `level` is the level or sub-band of the wavelet term, and `qfactor` is a user defined value which controls the quality. Notice that when `qfactor` equals 0, this new factor is always equal to 1 and has no effect. `level` is 0 for the scaling function (the one average term that remains when the transform is complete) and increases with higher frequency details. Note that you often have to special case the scaling function because its defined to be at level 0, which breaks some formulas (like the one above).
Feel free to use any quantization scheme that works, but this is a good starting point. Other schemes may give more intuitive control over quality. This scheme is by no means the best, but is simple and easy to implement. Another example of a simple to implement quantization scheme is thresholding: Use the `qfactor` parameter to control which terms are dropped to .5 (which is equivalent to 0 since we made the adjustment when calculating differences).

```
if(abs(p_i - .5) < qfactor)
    output_i = .5
else
    output_i = p_i
```

This behaves very differently from the other method described above. Also note that you don't necessarily want your function to behave linearly with `qfactor`, you may want to use powers of `qfactor` to get more intuitive control of quality. For example, in this thresholding scheme to get more even changes in quality you probably want to compare to `qfactor` to the third or fourth power. The exact image you get will depend on your quantization scheme, but it should look something like the one given in Fig. 1c with a high quality factor. Details will be visible in the higher subbands, but much of the image will be some shade of gray. As quality goes down, less detail should be visible.

2. **Reverse Quantization** To reconstruct the image from the quantized data, you need to Undo the qunatization step on the compressed data to get back an approximation of the floating point data before you apply the inverse 2D Haar transform. In this step you just need to reverse the quantization step you did above. The two functions should look very similar.

**Step 5: Quality of the Image Reconstruction** Now, play with different quality parameters and see how the resulting image(reconstructed) changes. There are many ways to measure the quality of the image reconstruction. We ask you to output two measurements.

1. Compute the PSNR (peak signal-to-noise ratio) of your reconstructed image with respect to the original image
2. Create a difference image of your reconstructed image with respect to the original image.

For the demo, you need to have two plots.

1. A graph of PSNR values vs. entropy for the `Monalisa.png`
2. A graph of PSNR values vs. entropy for one of the images you choose.

Take enough samples to get a good curve. How would one interpret this graph?

1. **PSNR** Given a grayscale image $I(x, y)$ and its reconstruction $I'(x, y)$, the PSNR is given by the ratio between the maximum squared intensity of the signal and the mean squared error (MSE). PSNR is reported in decibels (dB), so the full formula is

$$PSNR = 10 \log_{10} \left( \frac{I_{max}^2}{MSE} \right) \tag{7}$$

$$MSE = \frac{1}{wh} \sum_{x=0}^{w-1} \sum_{y=0}^{h-1} (I'(x, y) - I(x, y))^2 \tag{8}$$

You may use the luminance equation $Y = .299R + .587G + .114B$ (where $R$, $G$, and $B$ are the values of the red, green, and blue channels respectively) to compute the grayscale image.

2. **Difference Image** Another way to show error between the reconstructed image and the original image is to create a difference image between the original and reconstructed image.

```
DifferenceImage(i,j) = c * (Original(i,j) - Reconstructed(i,j)) + padding;
```

You can adjust `c` and the padding to scale and center the difference, so that the differences are clearer. A typical value for padding is 128 (gray). Another approach might be to show the absolute error

```
DifferenceImage(i,j) = abs(Original(i,j) - Reconstructed(i,j)) * c;
```

3. **Entropy** To compare the average number of bits we need to encode the input image and the transformed-quantized image. We calculate the entropy for both images. Each channel of a pixel on an image can take value from 0 - 255. You can count the number of pixel data (r, g, b) falling in each bin `Bin[0]...Bin[255]`. The distribution of the pixel values can be calculated by normalizing the bin counts with `3*w*h`.
Given a distribution, we can estimate the entropy using Eq. 9.

$$H(X) = -\sum_{i=0}^{255} p(x_i) \log_2 p(x_i) \tag{9}$$

   (a) Note that you should handle the case where `p(x)` equals 0.
   (b) In this assignment, you can ignore all spatial correlation between pixels.

## Demo and Grading

**Demo:** Try your program on different images. Save at least one screenshot. Comment on the compression behavior of the images during your demo. You also need to save two plots of PSNR vs. entropy at varying levels of quantization. Take enough samples to get a good curve. How would one interpret these graphs?

1. A graph of PSNR values vs. entropy for the `Monalisa.png`
2. A graph of PSNR values vs. entropy for one of the images you choose.

**Grading:** This assignment will be graded on a scale of 1-4. The values break down as:

1. Correct answers to questions *(1 point)*
2. Forward transform and inverse transform *(1 point)*
3. Quantization, transformed-quantized image, reverse quantization, and reconstructed image *(1 point)*
4. Entropy, PSNR, difference image, and plots *(1 point)*

## Hints and Tips

- Because we have positive and negative values the quantized values will be centered around 128. This will make your quantized image gray in most places. Note, however, that this doesn't affect compression since an entropy coder isn't affected by which symbols are most probable.
- Note the data types of the inputs and outputs of the various steps and be careful about the range of values you output.
- It can sometimes be hard to find errors in your code because of the repeated application of the functions. If you are having trouble debugging your code, try forcing your loop to perform only one step of the 2D transform. Once the quantized image looks right after one step, work on getting multiple steps to work.

- Remember you need to update your Makefiles or project files to point to the library files and include directories that you need. We want you to go through the process of setting up a working GLUT application on you system, but this will take some tinkering with your project setup.
- Getting `libST` to work in your XCode or Visual Studio project can be a bit tricky. You can grab `libST` from any of your previous assignments. Instructions are available for adding libst to your project in Visual Studio 2010 and Xcode 4