# PGF
## A new progressive file format for lossy and lossless image compression

**Christoph Stamm**
ETH Zurich, Institute of Theoretical Computer Science
ETH Zentrum, 8092 Zurich, Switzerland
stamm@inf.ethz.ch, http://www.inf.ethz.ch/personal/stamm/

**ABSTRACT**

We present a new image file format, called Progressive Graphics File (PGF), which is based on a discrete wavelet transform with progressive coding features. We show all steps of a transform based coder in detail and discuss some important aspects of our careful implementation. PGF can be used for lossless and lossy compression. It performs best for natural images and aerial ortho-photos. For these types of images it shows in its lossy compression mode a better compression efficiency than JPEG. This efficiency gain is almost for free, because the encoding and decoding times are only marginally longer. We also compare PGF with JPEG 2000 and show that JPEG 2000 is about ten times slower than PGF. In its lossless compression mode PGF has a slightly worse compression efficiency than JPEG 2000, but a clearly better compression efficiency than JPEG-LS and PNG. If both, compression efficiency and runtime, is important, then PGF is the best of the tested algorithms for compression of natural images and aerial photos.

**Keywords:** still image file format, lossy/lossless image compression, progressive coding, discrete wavelet transform.

## 1 Introduction

There are several dozens of different image file formats. Some of them use compression techniques while others do not. Some formats support only lossy while others also allow lossless compression. Some compression techniques are more useful for images of natural scenes rather than computer generated or artificial images. While stopping here enumerating the different features the question comes up: why does not exist a single image file format which supports all these features together with additional functionality like progressive decoding, region of interest coding, rate control, error resilience and so on? The answer is, there is a format providing all these features: JPEG 2000.

JPEG 2000 will be the next ISO/ITU-T standard for still image coding [SEA+00]. It is thought as a complement to the current JPEG standards. It should be used for low bit-rate compression and progressive transmission. It defines in its Part I the core system and in Part II various extensions for specific applications. It is based on the discrete wavelet transform (DWT), scalar quantization, context modeling, arithmetic coding and post-compression rate allocation. JPEG 2000 shows a very good rate-distortion performance (about 7% better than JPEG). Unfortunately, all the different implementations of the core system we have tested show a very bad encoding and decoding time compared to JPEG (about eight times slower). This means, JPEG 2000 is not a good solution if coding time is important or even crucial.

There are many applications where still image encoding or decoding time is important or crucial. For example, in professional digital photography a very fast image encoding is necessary if a series of pictures is taken in a short time and if compression is used. Compression is often used to store more digital images on a flash-memory card of a camera. Another example is a terrain explorer or a flight simulator with dynamic scene management [Paj98, Sta01], where aerial images are mapped onto the terrain to enhance the realistic appearance [Spu00]. In such a terrain explorer not only a very fast image decoding is important, but also a progressive loading strategy helps in reducing the amount of main memory needed. A third example is the transfer of large images over a channel with a low bandwidth relative to the amount of image data, e.g., transferring large images through the Internet. In this example progressive decoding is usually more important than fast decoding, because the transferred data should be as small as possible. Of course, in case of an image gallery fast decoding is still an obvious advantage.

For progressive image refinement methods with compression there is a trade-off between compression ratio, image quality, and compression/decompression time. For the same image quality methods with a lesser compression ratio tend to be faster. Most of the approaches try to achieve a maximum image quality for a given compression ratio. We call these approaches *quality driven*. In contrast, we are mainly interested in approaches with short decompression time and reasonable quality. We call these approaches *speed driven*.

In the following sections we discuss a new speed driven progressive image format with scalable resolution, called *Progressive Graphics File* (PGF)[1]. This image format serves very fast progressive refinement and achieves for the same compression ratios an image quality between JPEG and JPEG 2000.

The next section is a detailed description of our new image file format. Section 3 explains the comparison methodology employed in the results shown in Section 4. General conclusions are drawn in Section 5.

In the following we assume you are familiar with the one dimensional discrete wavelet transform (DWT). If not, you may read for example [FCD+95] or [Gra95].

[1] Progressive Graphics File (PGF) is a trademark of xeraina GmbH, Zurich, Switzerland. Internet: http://www.xeraina.ch

# 2 Progressive Graphics File (PGF)

PGF provides lossless and lossy compression. It is based on a fast, reversible integer DWT. Due to the hierarchical structure of the DWT a progressive refinement process with scalable resolution can be integrated naturally. The abandonment of floating point computations results in a crucial speedup, with an often negligible loss of image quality. The construction and reconstruction process chains are schematically illustrated in Fig. 1. Compression techniques are not much effective when applied to the original image, but can lead to reasonable performance when applied to quantized wavelet coefficients.

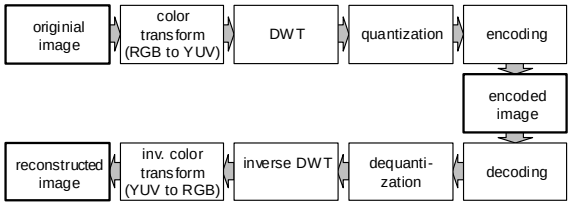


Fig. 1: Steps of a transform based coder and decoder.

## 2.1 Color Transform

We assume that the input of our process chain is a colored bitmap in *RGB* (Red, Green, Blue) format. In case of a grayscale bitmap we omit the first step, the color transform.

Usually the three channels of a natural RGB image contain both spatial and chromatic redundancy. An appropriate channel transformation can result in reduced redundancy and therefore, in a bundling of the energy in one channel. This energy channel is usually called the *achromatic* and the two remaining the *chromatic* channels. It is well-known that the human eye is more light than color sensitive. So, the resolution of the chromatic channels can be reduced without a large visual quality loss and hence a better compression ratio is possible.

Several popular color transforms of this type are known: e.g. *YIQ*, *YUV*, and *YCrCb*, where *Y* always denotes the achromatic channel. In both the *YUV* and the *YCrCB* color space the achromatic channel collects between eighty and ninety percent of the total energy. There are color transforms with even better energy bundling (> 99%) but also higher complexity [WGZ97].

Because PGF should be speed driven we are mainly interested in a simple integer approach. The following integer version of the *RGB*-*YUV* transform is used in PGF. The forward color channel transform is defined for each pixel as follows

$$\begin{aligned} Y &= \left\lfloor \frac{R+2G+B}{4} \right\rfloor - 2^{n-1} \\ U &= R - G \\ V &= B - G \end{aligned} \tag{1}$$

and the backward transform is defined as

$$\begin{aligned} G &= Y - \left\lfloor \frac{U+V}{4} \right\rfloor + 2^{n-1} \\ R &= U + G \\ B &= V + G, \end{aligned} \tag{2}$$

where $n$ denotes the number of bits in each $R$, $G$, and $B$ channel. While $Y$ occupies also $n$ bits, the chromatic channels $U$ and $V$ need in general $n$+1 bits. The integer implementation is very fast, because only addition and shifting operations are used.

In the following steps of our process chain the color transformed channels are used instead of the original image channels.

## 2.2 Discrete Wavelet Transform

The second step in our coder chain is a tensor product DWT. For each (color transformed) channel of an image we separately apply a tensor product DWT. In the rest of this paper we only discuss the appliance on one channel. This is enough, because we use the same DWT for all channels.

A two dimensional tensor product DWT is based on a one dimensional and ends up with a pyramid of wavelet transform coefficients. In each transform step we apply the one dimensional transform to the rows and columns of a matrix and downsample the output by 2 (see also Fig. 2, on page 4). In the beginning, the matrix is equal to the channel of the image. After one step, one ends up with four sub-bands: one average image *LL*, and three detail images *LH*, *HL*, and *HH*, where *L* denotes a low-pass and *H* a high-pass filter step. These four sub-bands together form a level. The next wavelet transform step does the same decomposition on the *LL* band of the last level. We continue while both the width and the height of the *LL* band are larger than two times the length of the larger filter. At the end we get a pyramid of levels, where each level is almost a quarter in size of the underlying level. The lowest level containing the original channel in its *LL* sub-band is called level 0. The three other sub-bands on level 0 are unused.

The crucial two points in our DWT are the choice of an appropriate wavelet filter set and the correct and careful integer implementation, which preserves the precision of the wavelet coefficients.

In [VBL95] 4300 biorthogonal wavelet filter banks for image compression have been tested according their impulse and step response and their average peak signal noise ratio (PSNR, see also Subsection 3.1) over eight test images using a eight bit quantization and a 16:1 compression. In case of an integer implementation of the transform, the possibility to use a small number of correct integer coefficients is important. For aerial ortho-photos used in flight simulators a high impulse response peak may be good, because a lot of artificial reference dots and crosses are introduced into the image in order to indicate absolute location. According to these criteria we have chosen a 5/3 filter set, where five and three denotes the length of the low- and high-pass filter, respectively. The coefficients of the filters are $k(-1, 2, 6, 2, -1)$ and $k(-2, 4, -2)$, with $k = 1/(4\sqrt{2})$. Another correct integer implementation with precision preservation based on a similar filter set is given in [CF97].

Our integer implementation uses the fact that the application of the factor of $\sqrt{2}$ in $k$ can be replaced by a normalization operation at the end of the transformation if the low-pass filter is divided by $\sqrt{2}$ and the high-pass filter is multiplied by $\sqrt{2}$. This normalization operation uses only division by 2 and multiplication by 2 and is therefore very fast in a integer implementation. Even better, the normalization can

be done at the same time as the quantization (cf. Subsection 2.3). We only discuss the exact transform for one level decomposition and reconstruction and only for a one dimensional signal. The extension to two dimensions is immediate as the rows and columns can be treated into a sequence of one dimensional signals. For the following algorithm, assume that $x_i$ is the original signal where $i \in [0, N-1]$ indicates a particular point in the signal of length $N$. Let $l_i$ and $h_i$ be the low-pass and the high-pass sub-sampled outputs, respectively. Further, let $M$ be the have length equal to $\lceil N/2 \rceil - 1$ and $c_1$ and $c_2$ are small integer constants.

The forward transform is computed according to Equation 3 and the corresponding reconstruction (inverse transform) is computed according to Equation 4.

In theory, both filters are first applied to the rows of a matrix and then to the columns of the result. In such an implementation with a large image, the application of the filters on the rows needs only the fraction of time it needs on the columns if the image is stored in rows. This is not very remarkable, because CPU caches use the principle of locality in space and time, and this locality is only given in accessing memory locations on a small number of rows. It is important to be aware of this and to implement the filter application in a manner which maximizes space and time locality. Therefore, we first filter only $r$ rows at a time, then apply the column filters on the result, and continue with the next rows, where $r$ is the minimum number of rows needed to apply the column filters on the result. This concept helps to drastically increase the space and time locality and hence to reduce the transformation time.

$$\begin{cases} h_0 = x_1 - \left\lfloor \dfrac{x_0 + x_2 + c_1}{2} \right\rfloor \\ l_0 = x_0 + \left\lfloor \dfrac{h_0 + c_1}{2} \right\rfloor \\ \left.\begin{aligned} h_k &= x_{2k+1} - \left\lfloor \dfrac{x_{2k} + x_{2k+2} + c_1}{2} \right\rfloor \\ l_k &= x_{2k} + \left\lfloor \dfrac{h_{k-1} + h_k + c_2}{4} \right\rfloor \end{aligned} \right\} k = 1,\ldots, M-1 \\ l_M = x_{N-1} + \left\lfloor \dfrac{h_{M-1} + c_1}{2} \right\rfloor, N \text{ is odd} \\ \left.\begin{aligned} h_M &= x_{N-1} - x_{N-2} \\ l_M &= x_{N-2} + \left\lfloor \dfrac{h_{M-1} + h_M + c_2}{4} \right\rfloor \end{aligned} \right\} N \text{ is even} \end{cases} \quad (3)$$

$$\begin{cases} x_0 = l_0 - \left\lfloor \dfrac{h_0 + c_1}{2} \right\rfloor \\ \left.\begin{aligned} x_{2k} &= l_k - \left\lfloor \dfrac{h_{k-1} + h_k + c_2}{4} \right\rfloor \\ x_{2k-1} &= h_{k-1} + \left\lfloor \dfrac{x_{2k-2} + x_{2k} + c_1}{2} \right\rfloor \end{aligned} \right\} k = 1,\ldots, M-1 \\ \left.\begin{aligned} x_{N-1} &= l_M - \left\lfloor \dfrac{h_{M-1} + c_1}{2} \right\rfloor \\ x_{N-2} &= h_{M-1} + \left\lfloor \dfrac{x_{N-3} + x_{N-1} + c_1}{2} \right\rfloor \end{aligned} \right\} N \text{ is odd} \\ x_{N-1} = h_M + x_{N-2}, N \text{ is even.} \end{cases} \quad (4)$$

## 2.3 Quantization

The goal of the quantization step in our coder chain is to reduce the information needed to store the image. This is the only step that introduces information loss. One can distinguish at least two kinds of quantization: vector and scalar. Vector quantization is in general more powerful than scalar quantization. In case of vector quantization, one replaces a group of coefficients (a vector) with one symbol. The key is to find the right way of grouping the coefficients such that as few symbols as possible are needed. For more details on vector quantization in combination with wavelets we refer to [ABMD92].

In case of scalar quantization, one divides the real or in our case the integer axis in a number of non-overlapping intervals, each corresponding to a symbol $s_i$. Each coefficient is now replaced by the symbol $s_i$ associated with the interval to which it belongs. The intervals and symbols are generally kept in a quantization table. An even simpler form of a scalar quantization is a uniform quantization with fixed interval length. In this form it is not even necessary to store a quantization table. Storing the interval length and the range is just enough, but the missing adaptivity in the uniform scalar quantization could lead to worse image quality. Sometimes, the scalar quantization is combined with a threshold, called dead zone. The idea of that threshold is to get a larger interval in which all wavelet transform coefficients are set to zero and therefore, to reduce a large number of coefficients and to produce a sparse matrix. It is important to omit the threshold in the quantization step of the last *LL* sub-band, because quantization errors in this sub-band lead to very poor image quality.

In PGF we use a uniform scalar quantization with dead zone. The interval length is restricted to powers of two. This makes it simple to combine the quantization with the normalization factor from the modified filter coefficients. This very simple quantization maximizes speed and minimizes storage, but at the same time it reduces the number of possible compression rates if quantization is the only source of information loss.

The dequantization step in the reconstruction process is the inverse of the quantization. Of course, we can reconstruct the exact values of the coefficients only if the interval length is equal to one. In all the other cases we reconstruct the wavelet coefficients incorrectly with the lower boundary of their interval.

## 2.4 Coding

The last step in our encoder chain is the encoding of the quantized wavelet transform coefficients into a bitstream. The problem in this step is finding a fast, storage efficient, and reversible method. Usually, the coding phase is a sequence of several different steps containing at least a reordering and a compression step. The goal of the reordering step is to cluster the wavelet coefficients in such a way that the compression step is more efficient.

In PGF we use a progressive wavelet coder (PWC) very similar to the coder presented in [Mal99]. PWC is based on progressive image coding, in which the bitstream is embedded, that is, representations of the image at any rate up to the encoding rate can be obtained simply by keeping the bitstream prefix corresponding to a desired rate. Embedded

encoding can be achieved simply by applying the well-known bit-plane encoding technique [SB66] to the scalar-quantized wavelet coefficients. The most significant bit-planes naturally contain many zeros, and therefore can be compressed without loss via entropy coders such as run-length coders. Since the reordering step influences the efficiency of the following compression step, it has to be chosen carefully and in knowledge of the type of the following compression step. In our PWC coder we use bit-plane coding of fixed size macroblocks.

In the following we describe both the reordering and compression step of the encoder. We omit the discussion of the decoder, because its functioning is even simpler and more or less inverse to the encoder.

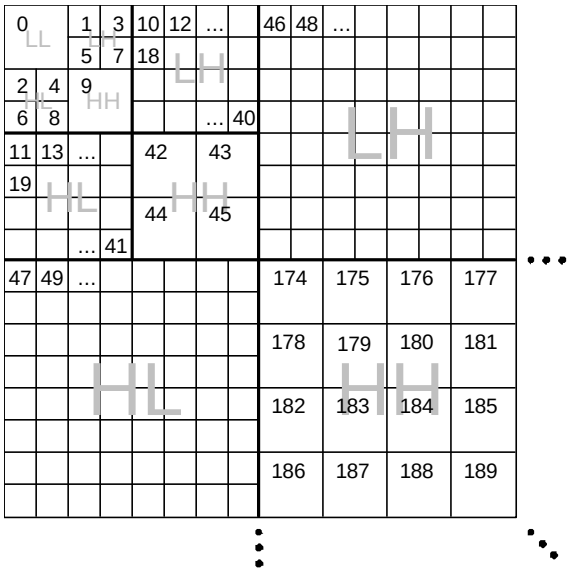


Fig. 2: Scanning order for defining blocks of wavelet coefficients.

### 2.4.1 Reordering Step

Bit-plane encoding is more efficient if we reorder the wavelet coefficients in such a way that coefficients with small absolute values tend to get clustered together. That translates into longer runs of zeros in the bit-planes, which can be encoded at lower bit rates. An efficient algorithm for achieving such clustering is the embedded zero tree coder [Sha93]. A similar technique to zero trees is used in the set partitioning in hierarchical trees (SPIHT) coder [SP96]. The SPIHT coder is very efficient in clustering zero-valued coefficients at any particular bit-plane; it attains very good compression results even without entropy encoding of the bit-plane symbols. SPIHT is one of the most efficient image compression algorithms reported to date. The PWC in PGF uses also clustering of the coefficients, but only in a simpler, data-independent and therefore faster way.

In a first step, we divide each sub-band into rectangular blocks: the *LL* and *HH* sub-bands into blocks of size 8×8 and the *LH* and *HL* sub-band into blocks of size 4×4. This partitioning scheme is depicted in Fig. 2. In a second step, we collect these blocks on each pyramid level, starting at the top and ending at level 1. We start with the blocks of the *LL* sub-band of the topmost level. Then on each level, first we alternately collect the blocks of the *LH* and the *HL* sub-band and continue with the blocks of the *HH* sub-band.

The reason for the alternate visiting of the *LH* and *HL* wavelet coefficients within the same level is simple. Assuming the original image has a particular feature at some spatial location, it is likely that clusters of both the *LH* and *HL* sub-bands, corresponding to that location, will have values of the same order. Therefore, by ensuring that pairs of blocks from the *LH* and *HL* sub-bands corresponding to the same spatial location appear contiguously in a macroblock, we are more likely to create clusters of similar values.

An obvious simpler clustering scheme with block sizes set to 1×1 does not perform in the same manner.

In the last step, we visit each collected block in the same order as collected and we write all coefficients of a block into a macroblock of fixed size $L$. $L$ is usually set to a power of two, for example 4096.

### 2.4.2 Compression Step

In our compression step we compress and encode each macroblock of the previous reordering step independently. The output of this step is usually written into a file. We use an adaptive run-length/Rice (RLR) coder [Lan83] to encode the non-zero bit-planes of a macroblock. Any efficient coder for asymmetric binary sources would suffice. For instance, adaptive arithmetic coding (AC) can be used instead of the adaptive RLR coder. The RLR coder is used, because of its simple implementation and its low time complexity. The RLR coder with parameter $k$ (logarithmic length of a run of zeros) is also known as the elementary Golomb code of order $2^k$ [OWS98]. In practice the RLR coder is very close to being an optimal variable-to-variable length coder [Fab92].

In our PWC we use the adaptive bit-plane encoding technique described in [Mal99] with minor changes. To see the difference we shortly summarize the original technique.

Suppose we start encoding a bit-plane $v = 0$, the most significant bit-plane, and proceed with increasing $v$, towards the least significant bit-plane. The algorithm works as follows:

1. Start with a macroblock of coefficients $c_i$ and define the significance flag vector $z$ such that $z_i = 0$ for all $i$. Set $v = 0$.
2. Let $b_i$ be the $v$-th bit of $|c_i|$. Break the set $\{b_i\}$ into two sets: $B_S = \{b_i \mid z_i = 0\}$ and $B_R = \{b_i \mid z_i = 1\}$.
3. Encode the sequence of bits in $B_S$ by a RLR coder and append the output to the bitstream.
4. Append the sequence of bits in $B_R$ to the bitstream.
5. Set $z_i = 1$ for all $i$ such that $b_i \in B_S$ and $b_i = 1$.
6. If the last bit-plane has not been coded yet, increase $v$ and return to Step 2.

In Step 3, the sequence of bits in $B_S$ has often long runs of zeros, specially in the initial bit-planes. We encode them with the RLR coder defined in Table 1.

| Codeword | Input bit sequence |
|---|---|
| 0 | Run of $2^k$ zeros |
| 1 $d$ 0 | Run of $d < 2^k$ zeros followed by a 1, $c_i \geq 0$ |
| 1 $d$ 1 | Run of $d < 2^k$ zeros followed by a 1, $c_i < 0$ |

Table 1: RLR coder for binary sources with parameter $k$.

The optimal value of the parameter $k$ depends on the probability that $c_i = 0$. The higher the probability the larger we should set $k$. Since we do not know the optimal value of $k$, we use an adaptive strategy. We start with $k = 0$ and adapt $k$ in a backward fashion, increasing it every time we emit a codeword '0', and decreasing it every time we emit a codeword starting with a '1'.

Until here, our compression step is identical to the corresponding coding step in [Mal99]. The difference is described in the following paragraph.

We already mentioned that bit-planes with a small $v$ have long runs of zeros and therefore a good run-length performance. Unfortunately, the performance of the RLR coder decreases with increasing $v$ and can even be negative, which means that the input bit sequence is shorter than the corresponding output of the RLR coder. The input sequence '01001' is a simple example with a negative RLR performance. The corresponding output is '010s011s', where $s$ is a substitute for a sign bit. The output length is 8 and the input length is 5+2, where 2 is the length of the number of ones in the input. If this happens, then we throw the output away and copy the input bit sequence as it is to the output. To specify if run-length coding has been used for the sequence of bits in $B_S$ or not, we add an additional flag in front of the corresponding output. In the decoder we read this flag and know immediately how we should interpret the bitstream. With this additional step we decrease the upper bound for the output length of a bit-plane.

# 3 Comparison Methodology

A major concern in coding techniques is the compression efficiency, but it is not the only factor that determines the choice of a particular algorithm for an application. Most applications also require a fast runtime and other features in a coding algorithm than simple compression efficiency. This is often referred to as functionalities. Examples of such functionalities are progressive decoding, or ability to distribute quality in a non-uniform fashion across the image. In Section 4 we report on trade-off between compression efficiency and runtime.

## 3.1 Compression Efficiency

Compression efficiency is measured for lossless and lossy compression. For lossless coding it is simply measured by the achieved compression ratio for each one of the test images. For lossy coding the *root mean square error* (RMSE), defined as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_i - q_i)^2} \qquad (5)$$

is used, as well as the corresponding *peak signal to noise ratio* (PSNR in dB), defined as

$$PSNR = 20\log_{10}\frac{\max_i|p_i|}{RMSE}, \qquad (6)$$

where $n$ is the number of pixels, $p_i$ are the pixels in the original and $q_i$ are the corresponding pixels in the compressed image. For an image with more than one channel (e.g. RGB image), we define the PSNR as the average over all PSNRs of the separate channels. A PSNR of 30 dB corresponds to a low quality image, while 50 dB means a visually almost perfect reconstruction. Although RMSE and PSNR are known to not always faithfully represent visual quality, they are the most established and well-known measure that works reasonably well across a wide range of compression ratios.

## 3.2 Runtime

Runtime is only a rough indication of algorithmic complexity. At least it is simple to measure, since evaluating complexity of an algorithm or an application is a more difficult issue. Complexity means different things for different applications. It can be asymptotical time and space complexity, total working memory, the used CPU instruction set, number of CPU cycles, number of hardware gates, etc. Furthermore, some of these numbers are very dependent on optimization and other factors of the different implementations.

We provide the runtimes of different file formats on a Windows 2000 based PC laptop with a 1 GHz Mobile Pentium™ III processor, 256 KByte cache memory and 384 MByte of RAM. All tested file formats, except WinZip, have been written in C or C++ and have been compiled with Microsoft Visual C++, version 6, with 'Maximize Speed' enabled.

The measured encoding times contain the steps needed to write an target image, including closing the file but excluding the time needed to open and read the source image. The decoding times are measured before the image has been opened and after the red, green, and blue pixels have been written to a memory buffer.

# 4 Tests and Results

To prove the quality of our PGF we evaluate several different image file formats and PGF with two different test sets. We test compression efficiency and runtime in lossless and lossy compression mode. The competitive file formats depend on lossless or lossy compression mode. In lossless mode we test PGF against WinZip[2], JPEG 2000[3], JPEG-LS[4], and PNG[5]. In lossy mode we test PGF against JPEG[6] and JPEG 2000.

The DWT of JPEG 2000 is dyadic and can be performed with either a reversible Le Gall 5/3 taps filter, which provides for lossless coding, or a non-reversible biorthogo-

[2] WinZip, version 6.3 SR-1, is a registered trademark of Nico Mak Computing, Inc.

[3] JPEG 2000, implementation of the JasPer project. The JasPer Project is a collaborative effort between Image Power, Inc. and the University of British Columbia. The objective of this project is to develop a software-based reference implementation of the codec specified in the JPEG-2000 Part-1 standard (i.e., ISO/IEC 15444-1). This software has also been submitted to the ISO for inclusion in the JPEG-2000 Part-5 standard (as an official reference implementation).; Internet: http://www.ece.ubc.ca/~mdadams/jasper/

[4] JPEG-LS, version 2.2, implementation of SPMG group of the University of British Columbia; Internet: http://spmg.ece.ubc.ca/

[5] Portable Network Graphics (PNG) is a W3C recommendation for coding still images which has been elaborated as a patent free replacement of GIF; the libpng implementation, version 1.0.12; Internet: ftp://ftp.uu.net/graphics/png/

[6] JPEG, implementation of the Independent JPEG Group, version 6b; Internet: http://www.ijg.org/

nal 9/7 filter, which provides for higher compression but does not do lossless compression. The quantization is an embedded scalar approach with threshold and is independent for each sub-band. Each sub-band is entropy coded using context modeling and bit-plane arithmetic coding. The generated code-stream is parseable and can be resolution, quality, position or component progressive, or any combination thereof.

In our tests we only use the reversible 5/3 filter of JPEG 2000, because it is based on integers and is therefore clearly faster than the 9/7 filter which is based on floating point operations. For PNG the maximum compression setting is used, while for JPEG-LS the default options are chosen.

## 4.1 Test Sets

The first test set, called PGF test set, contains seven images covering various types of imagery. The images 'woman' (512×768), 'hibiscus' (768×512), 'houses' (768×512), and 'redbrush' (1024×960) are natural. The images 'woman', 'hibiscus', and 'houses' are identical with the images number four, seven, and eight of our second test set. The image 'compound' (1400×1050) is a screen shot consisting of text, charts and computer graphics, 'aerial' (1024×1024) is an aerial ortho-photo, and 'logo' (615×225) is a computer generated logo consisting of text. All these images have a depth of 24 bit per pixel.

The second test set, called Kodak test set, contains the first eight images of the Kodak true color test set (768×512)[7]. All images in the Kodak test set are natural and have a depth of 24 bit per pixel.

In order to make better comparisons with other methodologies, we also perform our technique on the black and white image 'Lena' (512×512, 8 bit per pixel).

## 4.2 Lossless Compression

Table 2 summarizes the lossless compression efficiency and Table 3 the coding times of the PGF test set. For WinZip we only provide average runtime values, because of missing source code we have to use an interactive testing procedure with runtimes measured by hand. All other values are measured in batch mode.

| | WinZip | JPEG-LS | JPEG 2000 | PNG | PGF |
|---|---|---|---|---|---|
| aerial | 1.352 | 2.073 | 2.383 | 1.944 | 2.314 |
| compound | 12.451 | 6.802 | 6.068 | 13.292 | 4.885 |
| hibiscus | 1.816 | 2.200 | 2.822 | 2.087 | 2.538 |
| houses | 1.241 | 1.518 | 2.155 | 1.500 | 1.965 |
| logo | 47.128 | 16.280 | 12.959 | 50.676 | 10.302 |
| redbrush | 2.433 | 4.041 | 4.494 | 3.564 | 3.931 |
| woman | 1.577 | 1.920 | 2.564 | 1.858 | 2.556 |
| **average** | **9.71** | **4.98** | **4.78** | **10.70** | **4.07** |

Table 2: Lossless compression ratios of the PGF test set.

In Table 2 it can be seen that in almost all cases the best compression ratio is obtained by JPEG 2000, followed by PGF, JPEG-LS, and PNG. This result is different to the result in [SEA+00], where the best performance for a similar test set has been reported for JPEG-LS. PGF performs between 0.5% (woman) and 21.3% (logo) worse than JPEG 2000. On average it is almost 15% worse. The two exceptions to the general trend are the 'compound' and the 'logo' images. Both images contain for the most part black text on a white background. For this type of images, JPEG-LS and in particular WinZip and PNG provide much larger compression ratios. However, in average PNG performs the best, which is also reported in [SEA+00].

These results show, that as far as lossless compression is concerned, PGF performs reasonably well on natural and aerial images. In specific types of images such as 'compound' and 'logo' PGF is outperformed by far in PNG.

| | WinZip | | JPEG-LS | | JPEG 2000 | | PNG | | PGF | |
|---|---|---|---|---|---|---|---|---|---|---|
| | enc | dec | enc | dec | enc | dec | enc | dec | enc | dec |
| a | | | 1.11 | 0.80 | 5.31 | 4.87 | 3.70 | 0.19 | 0.99 | 0.77 |
| c | | | 1.61 | 0.38 | 3.46 | 3.06 | 2.95 | 0.18 | 0.95 | 0.80 |
| hi | | | 0.69 | 0.30 | 1.45 | 1.29 | 1.77 | 0.10 | 0.35 | 0.27 |
| ho | | | 0.65 | 0.30 | 1.62 | 1.47 | 0.85 | 0.11 | 0.41 | 0.32 |
| l | | | 0.09 | 0.02 | 0.26 | 0.21 | 0.16 | 0.01 | 0.07 | 0.06 |
| r | | | 0.65 | 0.44 | 4.29 | 4.01 | 3.61 | 0.16 | 0.66 | 0.59 |
| w | | | 0.39 | 0.30 | 1.76 | 1.63 | 1.08 | 0.08 | 0.35 | 0.27 |
| **av** | **1.14** | **0.37** | **0.74** | **0.36** | **2.59** | **2.36** | **2.02** | **0.12** | **0.54** | **0.44** |

Table 3: Runtime of lossless compression of the PGF test set

Table 3 shows the encoding (enc) and decoding (dec) times (measured in seconds) for the same algorithms and images as in Table 2. JPEG 2000 and PGF are both symmetric algorithms, while WinZip, JPEG-LS and in particular PNG are asymmetric with a clearly shorter decoding than encoding time. JPEG 2000, the slowest in encoding and decoding, takes more than four times longer than PGF. This speed gain is due to the simpler coding phase of PGF. JPEG-LS is slightly slower than PGF during encoding, but slightly faster in decoding images. WinZip and PNG decode even more faster than JPEG-LS, but their encoding times are also worse. PGF seems to be the best compromise between encoding and decoding times.

Our PGF test set clearly shows that PGF in lossless mode is best suited for natural images and aerial ortho-photos. PGF is the only algorithm that encodes the three MByte large aerial ortho-photo in less than second without a real loss of compression efficiency. For this particular image the efficiency loss is less than three percent compared to the best. These results should be underlined with our second test set, the Kodak test set.

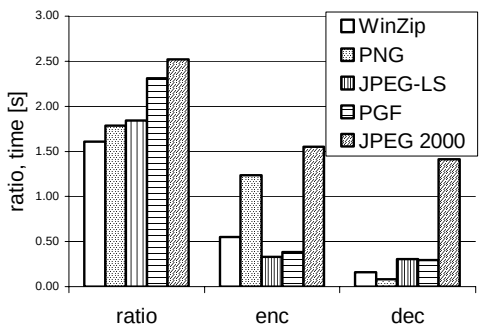


Fig. 3: Lossless compression results of the Kodak test set.

[7] The lossless true color Kodak test images in png format are available at: http://sqez.home.att.net/thumbs/Thumbnails.html

Fig. 3 shows the averages of the compression ratios (ratio), encoding (enc), and decoding (dec) times over all eight images. JPEG 2000 shows in this test set the best compression efficiency followed by PGF, JPEG-LS, PNG, and WinZip. In average PGF is eight percent worse than JPEG 2000. The fact that JPEG 2000 has a better lossless compression ratio than PGF does not surprise, because JPEG 2000 is more quality driven than PGF. However, it is remarkable that PGF is clearly better than JPEG-LS (+21%) and PNG (+23%) for natural images.

JPEG-LS shows in the Kodak test set also a symmetric encoding and decoding time behavior. Its encoding and decoding times are almost equal to PGF. Only PNG and WinZip can faster decode than PGF, but they also take longer than PGF to encode.

If both compression efficiency and runtime is important, then PGF is clearly the best of the tested algorithms for lossless compression of natural images and aerial ortho-photos.

In the third test we perform our lossless coder on the 'Lena' image. The compression ratio is 1.68 and the encoding and decoding takes 0.25 and 0.19 seconds, respectively.

## 4.3 Lossy Compression

Originally, PGF has been designed to quickly and progressively decode lossy compressed aerial images. A lossy compression mode has been preferred, because in an application like a terrain explorer texture data (e.g., aerial ortho-photos) is usually mid-mapped filtered and therefore lossy mapped onto the terrain surface. In addition, decoding lossy compressed images is usually faster than decoding lossless compressed images.

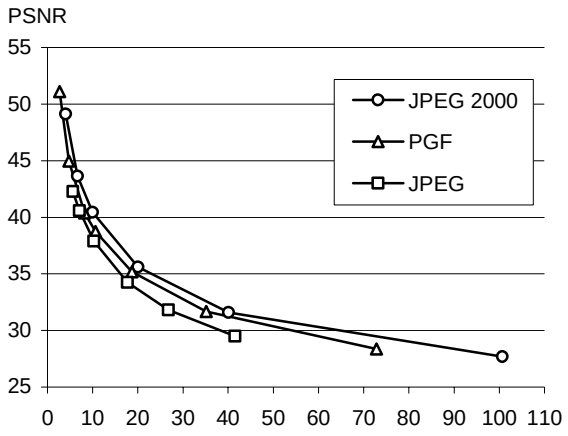


Fig. 4: PSNR of lossy compression in relation to compression ratio.

In the next test series we evaluate the lossy compression efficiency of PGF. One of the best competitors in this area is for sure JPEG 2000. Since JPEG 2000 has two different filters, we used the one with the better trade-off between compression efficiency and runtime. On our machine the 5/3 filter set has a better trade-off than the other. However, JPEG 2000 has in both cases a remarkable good compression efficiency for very high compression ratios but also a very poor encoding and decoding speed. The other competitor is JPEG. JPEG is one of the most popular image file formats. It is very fast and has a reasonably good compression efficiency for a wide range of compression ratios. The drawbacks of JPEG are the missing lossless compression and the often missing progressive decoding.

Fig. 4 depicts the average rate-distortion behavior for the images in the Kodak test set when fixed (i.e., non-progressive) lossy compression is used. The PSNR of PGF is on average 3% smaller than the PSNR of JPEG 2000, but 3% better than JPEG. These results are also qualitative valid for our PGF test set and they are characteristic for aerial ortho-photos and natural images.

Because of the design of PGF we already know that PGF does not reach the compression efficiency of JPEG 2000. However, we are interested in the trade-off between compression efficiency and runtime. To report this trade-off we show in Table 4 a comparison between JPEG 2000 and PGF and in Fig. 5 (on page 8) we show for the same test series as in Fig. 4 the corresponding average decoding times in relation to compression ratios.

Table 4 contains for seven different compression ratios (mean values over the compression ratios of the eight images of the Kodak test set) the corresponding average encoding and decoding times in relation to the average PSNR values. In case of PGF the encoding time is always slightly longer than the corresponding decoding time. The reason for that is that the actual encoding phase (cf. Subsection 2.4.2) takes slightly longer than the corresponding decoding phase. For six of seven ratios the PSNR difference between JPEG 2000 and PGF is within 3% of the PSNR of JPEG 2000. Only in the first row is the difference larger (21%), but because a PSNR of 50 corresponds to an almost perfect image quality the large PSNR difference corresponds with an almost undiscoverable visual difference. The price they pay in JPEG 2000 for the 3% more PSNR is very high. The creation of a PGF is five to twenty times faster than the creation of a corresponding JPEG 2000 file, and the decoding of the created PGF is still five to ten times faster than the decoding of the JPEG 2000 file. This gain in speed is remarkable, especially in areas where time is more important than quality, maybe for instance in real-time computation.

| | JPEG 2000 5/3 | | | PGF | | |
|---|---|---|---|---|---|---|
| Ratio | enc | dec | PSNR | enc | dec | PSNR |
| 2.7 | 1.86 | 1.35 | 64.07 | 0.34 | 0.27 | 51.10 |
| 4.8 | 1.75 | 1.14 | 47.08 | 0.27 | 0.21 | 44.95 |
| 8.3 | 1.68 | 1.02 | 41.98 | 0.22 | 0.18 | 40.39 |
| 10.7 | 1.68 | 0.98 | 39.95 | 0.14 | 0.13 | 38.73 |
| 18.7 | 1.61 | 0.92 | 36.05 | 0.12 | 0.11 | 35.18 |
| 35.1 | 1.57 | 0.87 | 32.26 | 0.10 | 0.09 | 31.67 |
| 72.9 | 1.54 | 0.85 | 28.86 | 0.08 | 0.08 | 28.37 |

Table 4: Trade-off between quality and speed for the Kodak test set

In Fig. 5 we see that the price we pay in PGF for the 3% more PSNR than JPEG is low: for small compression ratios (< 9) decoding in PGF takes two times longer than JPEG and for higher compression ratios (> 30) it takes only ten percent longer than JPEG. These test results are characteristic for both natural images and aerial ortho-photos.

Again, in the third test series we only use the 'Lena' image. We run our lossy coder with six different quantization parameters and measure the PSNR in relation to the resulting compression ratios. The results (ratio: PSNR) are: (1.82:

53.20); (2.57: 45.35); (4.48: 39.85); (9.20: 36.06); (17.70: 33.29); (33.98: 30.58).

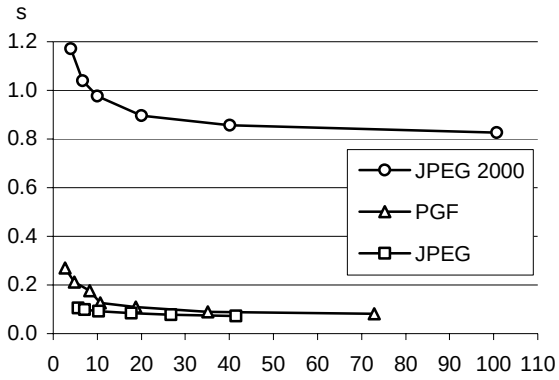


Fig. 5: Decoding time in relation to compression ratio

# 5 Conclusions

For our three dimensional terrain explorer we looked for an image file format that meets our demands: fast compression and decompression, lossy and maybe also lossless compression with good compression efficiency, and progressive image coding in terms of increasing resolution. Because we could not find an established image file format meeting all these demands, we designed and developed a new file format, called PGF, which is based on a discrete wavelet transform with progressive coding features.

We presented several changes to an earlier presented wavelet coders, like a new reordering scheme for wavelet coefficients or a new compression step during the encoding phase. Additionally, we carefully implemented all parts necessary for a working encoder and decoder, and we reported in several test series the efficiency of our algorithm.

PGF can be used for lossless and lossy compression. It performs best for natural images and aerial ortho-photos. For these types of images it shows in its lossy compression mode a three percent better compression efficiency than JPEG with only ten to hundred percent more encoding time. In contrast, JPEG 2000 achieves a seven percent better compression efficiency than JPEG, but has a ten times longer decoding time. This means, the price we pay in PGF for the additional feature of progressive coding is very low.

In lossless compression mode it also performs well for natural images and aerial ortho-photos. For these image types it has a eight percent worse compression efficiency than JPEG 2000, but a 21% better compression efficiency than JPEG-LS and 23% better than PNG. PGF is more than four times faster than JPEG 2000. Therefore, if both compression efficiency and runtime is important, then PGF is clearly the best of the tested algorithms for lossless compression of natural images and aerial ortho-photos.

# References


[ABMD92] M. Antonini, M. Barlaud, P. Mathieu, I. Daubechies. Image Coding using the Wavelet Transform. In *IEEE Transactions on Image Processing*, 1(2):205–220, 1992.

[CF97] H. Chao, P. Fisher. An Approach to Fast Integer Reversible Wavelet Transforms for Image Compression. CompSci, 1996.

[Fab92] F. Fabris. Variable-length to variable-length source coding: a greedy step-by-step algorithm. In *IEEE Trans. Inform. Theory*, 38: 1609–1617, 1992.

[FCD+95] A. Fournier, M. F. Cohen, T. D. DeRose, M. Lounsbery, L.-M. Reissell, P. Schröder, W. Sweldons. Wavelets and their Applications in Computer Graphics. *Course Notes*, SIGGRAPH ’95, 1995.

[Gra95] A. Graps. An Introduction to Wavelets. In *IEEE Computational Science and Engineering*, 2(2), Los Alamitos, 1995.

[Lan83] G. G. Langdon, Jr. An adaptive run-length encoding algorithm. In *IBM Tech. Discl. Bull.*, 26:3783–3785, 1983.

[Mal99] H. S. Malvar. Fast Progressive Wavelet Coding. *Proceedings IEEE DCC’99*, 1999.

[OWS98] E. Ordentlich, M. Weinberger, G. Seroussi. A low-complexity modeling approach for embedded coding of wavelet coefficients. In *Proceedings Data Compression Conference*, 408–417, Snowbird, Utah, 1998.

[Paj98] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings Visualization 98*, Los Alamitos. IEEE Computer Society Press, 1998.

[SB66] J. W. Shwartz, R. C. Baker. Bit-plane encoding: a technique for source encoding. In *IEEE Trans. Aerospace Electron. Syst.*, 2:385–392, 1966.

[SEA+00] S. Santa Cruz, T. Ebrahimi, J. Askelof, M. Larsson, C. Christopoulos. An analytical study of JPEG 2000 functionalities. In *Proceedings of SPIE of the 45th annual SPIE meeting, Applications of Digital Image Processing XXIII*, vol. 4115, 2000.

[Sha93] J. M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelet Coefficients. In *IEEE Transactions on Signal Processing*, 41(12): 3445–3462, 1993.

[SP96] A. Said, W. A. Pearlman. A new, fast, and different image codec based on set partitioning in hierarchical trees. In *IEEE Transactions on Circuits and Systems for Video Tech.*, 6:243–250, 1996.

[Spu00] R. Spuler. Progressive Texture. Diplomarbeit, Institute of Theoretical Computer Science, ETH Zurich, 2000.

[Sta01] C. Stamm. Algorithms and Software for Radio Signal Coverage Prediction in Terrains. Ph. D. thesis, Institute of Computer Science, ETH Zurich, 2001.

[VBL95] J. D. Villasenor, B. Belzer, J. Liao. Wavelet Filter Evaluation for Image Compression. *IEEE Transactions on Image Processing*, 1995.

[WGZ97] S. G. Wolf, R. Ginosar, Y. Y. Zeevi. Spatio-chromatic image enhancement based on a model of human visual information processing, 1997.