7-1-1995

An Object Oriented Implementation of Fractal Image Compression

Darrell Burkhead
Western Kentucky University

Follow this and additional works at: http://digitalcommons.wku.edu/theses
Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation
http://digitalcommons.wku.edu/theses/897

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
A Thesis

Presented to
the Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

Darrell Ray Burkhead

July 1995
AN OBJECT-ORIENTED IMPLEMENTATION OF FRACTAL IMAGE COMPRESSION

Date Recommended: April 24, 1994

Robert Crawford
Director of Thesis

Ali A. Khordost
Elva C. Cullinan

Elmer Gray
Director of Graduate Studies 7/17/95
Date
I would like to thank Dr. Crawford for his patience and help during this project. I would also like to thank everyone else who has encouraged me along the way. This paper is dedicated to the memory of Reba Burkhead.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. FRACTALS</td>
<td>2</td>
</tr>
<tr>
<td>3. OVERVIEW OF FIC</td>
<td>7</td>
</tr>
<tr>
<td>4. FIC IMPLEMENTATION</td>
<td>14</td>
</tr>
<tr>
<td>5. CONCLUSIONS</td>
<td>32</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>34</td>
</tr>
<tr>
<td>WORKS CITED</td>
<td>98</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Sierpinski Triangle</td>
<td>5</td>
</tr>
<tr>
<td>2. The FIC Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>3. Input &amp; Output of FIC</td>
<td>30</td>
</tr>
<tr>
<td>4. An FIC Object/Method Breakdown</td>
<td>32</td>
</tr>
</tbody>
</table>
The technique of Fractal Image Compression, although new, has been described in several ways. Thus far, all descriptions of this compression algorithm read by the author have been in procedural form. The purpose of this paper is to present the Fractal Image Compression algorithm in an object-oriented form and to point out the advantages of this organization.

The main advantages of taking an object-oriented approach to this problem are flexibility and maintainability. Different aspects of this algorithm are handled by different objects, thereby allowing for easy customization and testing of each part.

Another advantage of this approach is that the structure of objects serves as a tutorial for what the algorithm does. By examining the object structure, one can get a feel for the algorithm's operation without studying the underlying mathematics.
CHAPTER 1

INTRODUCTION

In this paper, the author discusses a relatively new technique of compression, Fractal Image Compression (FIC). Fractal refers to the conversion of an image into a fractal representation that occurs during FIC. Image refers to type of information being compressed; image data has been the traditional target of FIC due to certain “fractal” properties that images possess. Compression refers to the goal of FIC, to reduce the amount of “space” necessary to represent the image in question.

Before discussing Fractal Image Compression, some background concepts need to be reviewed. From that point, the discussion will continue with an overview of FIC itself: why and how it works. Finally, the discussion will conclude with an examination of an object-oriented implementation of FIC.
The term fractal has come to mean a great many things since it was coined by Benoit Mandelbrot in 1975 (Gleik 98). To some, it means a group of colorful and amazingly-detailed pictures. To others, it means the population-growth patterns of a species. To yet others, it means length of a coastline or the shape of a cloud. Common to these definitions are some properties that fractals are said to possess:

- self-similarity—fractals repeat some larger pattern in smaller details, as seen in a fingerprint.
- scalability—fractals retain the same amount of detail independent of the scale at which they are viewed. For example, a mountain demonstrates the same complexity whether viewed from miles away, from atop the mountain, or under a microscope.

These properties are found in images of real objects. As the research will show, it is the self-similarity present in image files that makes them amenable to FIC. Typical computer representations of images are decidedly not scalable. Bit-mapped computer representations of images\(^1\) are sampled at a fixed granularity. The resulting pixels become blobs of color when scaled. Images compressed by FIC inherit the scalability property; they exhibit the same amount of detail when scaled.

Thus far, FIC has been restricted to image data. Although it was designed for images, it could be applied other forms of information that are self-similar. Sound is such a form of information. The “scaling” problems with computer representations of sound are analogous to the problems with computer images. Sounds are sampled at a fixed granularity; the sounds are quantized at fixed

---

\(^1\) Vector representations of images are scalable. However, vector representations are typically used in CAD/CAM programs. They are not usually employed for real-world images, since many things in the real world cannot be accurately represented in terms of Euclidean-geometric shapes.
intervals. Scaling in terms of sound corresponds to playing back the sound at a higher rate of sampling. As with computer images, when computer sounds are scaled, they take on a "grainy" quality. Computer sounds compressed with FIC could be replayed at an arbitrary sampling rate with no decrease in sound quality.

The fractals that will be used in this paper are called Iterated Function System (IFS) fractals. An IFS is a finite set of functions from a metric space (a set with an associated concept of distance that is consistent\(^2\)) to the same metric space. Choosing \( \mathbb{R} \) with Euclidean distance \( d(x_0, x_1) = |x_0 - x_1| \) as the metric space, the following functions form an IFS:

\[
\begin{align*}
  f_1(x) &= x + 3 \\
  f_2(x) &= 2x - 4 \\
  f_3(x) &= x^2 + 2
\end{align*}
\]

The above definition of IFS covers a broad band of possibilities. For the purposes of this paper, it is necessary to restrict the examination to a subset of IFSes. This subset is called the hyperbolic iterated function systems. A hyperbolic IFS "consists of a complete metric space\(^3\) \((X, d)\) together with a finite set of contraction mappings \( w_i: X \to X \), with respective contractivity factors \( s_i \), for \( i = 1, 2, \ldots n \)" (Barnsley 82). A "contraction mapping" is a function that moves points closer. In terms of the definition, \( w_i: X \to X \) is a contraction mapping with contractivity factor \( s_i \) if, given any \( x, y \in X \),

\[
d(w_i(x), w_i(y)) \leq s_i \cdot d(x, y).
\]

The use of contractive maps is essential to producing fractals. Contractive maps are also called attractors, since they tend to attract the metric space to a fixed point in that metric space (Barnsley 76). For example, using \( \mathbb{R} \) with Euclidean distance as the metric space, \( f(x) = \frac{1}{2}x + \frac{1}{2} \) is an attractor whose fixed point is 1. The following properties of attractors are important to their use in IFS fractals:

\(^2\) Consistent in this case means that the following statements hold true for the set, call it \( X \), and the distance function, call it \( d: \forall a, b, c \in X, d(a, b) = d(b, a); a \neq b \Rightarrow 0 < d(a, b) < \infty; d(a, a) = 0; \text{ and } d(a, b) + d(b, c) \geq d(a, c). \)

\(^3\) \( X \) is a complete metric space \( \Leftrightarrow \) every Cauchy sequence \( \{x_n\}_{n=1}^\infty \) in \( X \) has a limit \( x \in X \)
• The fixed point of an attractor is not moved by the attractor.
• Given any point in the metric space, repeated applications of the attractor will eventually move that point to the fixed point. In other words, given a complete metric space \((X, d)\), an attractor \(f: X \to X\), and \(x \in X\), \(\lim_{n \to \infty} f^n(x) = x_f\), the fixed point.\(^4\)

To see the effects of a hyperbolic IFS, it is necessary to switch the frame of reference. Repeated applications of a contractive map will lead to a single, fixed point; thus, it makes sense to talk about a contractive map’s effect on points. Repeated applications of a hyperbolic IFS will lead to a single, fixed subset of the metric space; hence, the discussion of hyperbolic IFSes will be in terms of subsets of the metric space.

If \((X, d)\) is a complete metric space, \((\mathcal{H}(X), h(d))\) is also a metric space where \(\mathcal{H}(X)\) stands for the set of all compact subsets of \(X\) other than \(\emptyset\).\(^5\) The corresponding distance function \(h(d)\), called the Hausdorff distance function, is based on the distance function under \(X\). \(h(d)(A, B) = \max\{\min\{d(a, b) : b \in B\} : a \in A\}\). \(h(d)\) represents the maximum distance between any point in \(A\) and the set \(B\). \(h(d)\) is a consistent way to measure distance in \(\mathcal{H}(X)\),\(^6\) so \((\mathcal{H}(X), h(d))\) is a metric space.

Given a complete metric space \((X, d)\) and a hyperbolic IFS \(W = \{w_i : X \to X, i = 1, 2, \ldots n\}\), where the \(w_i\)'s are contractive maps, \(W: \mathcal{H}(X) \to \mathcal{H}(X)\) is defined by the following:

\[
\forall A \in \mathcal{H}(X), W(A) = \bigcup_{i=1}^{n} w_i(A)
\]

Applying \(W(A)\) results in a consensus of the component functions. One can prove that \(W\) is a contractive map, that is \(\forall A, B \in \mathcal{H}(X), h(d)(W(A), W(B)) \leq s \cdot h(d)(A, B)\) for some \(0 < s < 1\).\(^7\)

---

\(^4\) For a proof of these properties of contractive maps, see (Barnsley 75–76).

\(^5\) In topological terms, a compact subset of a space is a subset where “every open cover has a finite subcover.” FIC deals with images, so the metric space \((R^2, \text{Euclidean})\) (or some subset of \(R^2\)) will be used in most cases. In \((R^2, \text{Euclidean})\), a set is compact \(\iff\) it is closed and bounded—that is, \(\iff\) the set could be drawn.

\(^6\) \(\forall A, B, C \in \mathcal{H}(X), h(d)(A, B) = h(d)(B, A); 0 < h(d)(A, B) < \infty, \text{if } A \neq B; h(d)(A, A) = 0; \text{ and } h(d)(A, C) \leq h(d)(A, B) + h(d)(B, C).\)

\(^7\) For the proof of this result, see (Barnsley 80–81).
In the case of $W$, the “fixed point” of the contractive map, call it $A_f$, is a compact subset of $X$, a fractal. The properties of attractors stated before also apply to $W$:

- $W(A_f) = A_f$.
- Given any compact subset of $X$, repeated applications of the $W$ will eventually produce $A_f$.

Therefore, $\forall A \in \mathcal{H}(X) \lim_{n \to \infty} W^{\circ n}(A) = A_f$.

Using $\mathbb{R}^2$ with Euclidean distance as the metric space, the fixed point of the following hyperbolic IFS:

\[
\begin{align*}
    f_1(x, y) &= \left(\frac{1}{2}x + 1, \frac{1}{2}y + 1\right) \\
    f_2(x, y) &= \left(\frac{1}{2}x + 1, \frac{1}{2}y + 50\right) \\
    f_3(x, y) &= \left(\frac{1}{2}x + 50, \frac{1}{2}y + 50\right)
\end{align*}
\]

is called the Sierpinski triangle (Barnsley 86).

\[\text{Figure 1 : The Sierpinski Triangle}\]

The self-similarity of the Sierpinski triangle is immediately evident. Each of the three triangles that make up the main triangle is a Sierpinski triangle, and so on ad infinitum. The scalability of the Sierpinski triangle is closely related to its self-similarity; each subtriangle is a Sierpinski triangle with as much detail as the main triangle.

$\mathcal{H}(\mathbb{R}^2)$ serves as a model for the set of black-and-white images. Every hyperbolic IFS over $\mathcal{H}(\mathbb{R}^2)$ (with an associated metric) produces a fractal image. These fractal images all follow the definition in a similar manner, with the function being defined differently for grayscale and color images. In the case of grayscale images, the function becomes $f: [0, 1] \to [0, 1]$, where each pixel is mapped to its grayscale value. In the case of color images, the function becomes $f: [0, 1]^3 \to [0, 1]^3$, where each pixel is mapped to one of the many 3-dimensional forms of expressing color: Red, Green,
same general pattern as the Sierpinski Triangle; they all possess a noticeable degree of self-similarity that pervades the image. This high degree of self-similarity means that not every image can be produced by a hyperbolic IFS. Real-world images possess some degree of self-similarity. However, several self-similar patterns may be present and/or the self-similarity may be restricted to a part of the image. Therefore, a new construct is necessary to produce real-world images.

The building block of the “new construct” is called a local contraction map. A local contraction map contracts a subset of the metric space. Given a compact metric space \((X, d)\) and \(\emptyset \neq A \subset X\), \(w : A \to X\) is a local contraction map \(\exists s \geq 0 \leq s < 1, \forall x, y \in A, d(w(x), w(y)) \leq s \cdot d(x, y)\).

As a side note, the fact that \(w\)’s range is not restricted also means that \(w(w(x))\) does not necessarily make sense.

The new construct is called a local IFS. It is similar to a hyperbolic IFS, except that the contractive maps are replaced with local contractive maps. A local IFS consists of a compact metric space \((X, d)\) and local contractive maps \(w_i : A_i \to X\), for \(i = 1, 2, \ldots, n\), where the \(A_i\)’s are non-empty subsets of \(X\). Similarly to a hyperbolic IFS, if \(W = \{X : w_i : A_i \to X, i = 1, 2, \ldots, n\}\) is a local IFS, then \(W : S \to S\), where \(S\) denotes the set of subsets of \(X\), and \(W\) is defined by

\[
\forall B \in S, W(B) = \bigcup_{i=1}^{n} w_i(A_i \cap B)
\]

Like a hyperbolic IFS, applying a local IFS results in a consensus of the component functions. Each \(w_i\) maps the portion of \(B\) which lies within its range, \(A_i\).

Unlike a hyperbolic IFS, a local IFS does not necessarily have a fixed point. The action of a local IFS is harder to follow than in a hyperbolic IFS. Despite these problems with local IFSes, they have the versatility to represent real-world images. In essence, FIC consists of converting an image to the local IFS that will produce that image.

and Blue intensities (RGB), Hue, Saturation, and Value (HSV), etc.
Fractal Image Compression differs from entropy-encoding compression methods in several different ways:

- The search for redundancy is more holistic. The entire image is analyzed trying to match up self-similar parts of the image.
- It is potentially a “lossy” compression method, that is, the image produced by FIC might not exactly match the original. Some loss is acceptable in images as long as the result looks like the original.
- It is a two-step compression process. Once an image has been converted to a local IFS, the local IFS data is compressed with an entropy-encoding method.

The following figure shows the general steps involved in Fractal Image Compression:

![Figure 2: The FIC Algorithm](image)

9 Entropy-encoding compression methods try to remove redundancy on a character-by-character basis. Some examples include: run-length encoding and the LZW algorithm.
In this paper, the concern is with the results of $\Rightarrow$, the fractal image file. What happens after $\Rightarrow$ can either be integrated into the system or done "by hand" using a separate compression program. In either case, a discussion of entropy-encoding methods is beyond the scope of this paper.

Computer images have been typically represented as a matrix of fixed-sized dots, or pixels. Color information is stored for each of the pixels. By using entropy encoding, one can reduce the amount of space necessary to store an image. However, the resulting image data is still tied to the same matrix of pixels, that is, the same resolution.

An image file produced by FIC is no longer linked to the fixed-sized dots that originally described the image. FIC attempts to describe an image in terms of an underlying order. As a result of this translation, the color that a fractal image file says should be at location $(0, 1)$ may not exactly match up with the color from the original image. However, the color from the fractal image file should "look like" the original color, that is it should lie within an acceptable variance from the original. In addition, by using the same process as was used to get a color value for $(0, 1)$, one can produce a color value for $(\frac{1}{3}, \frac{1}{3})$.

The process of describing the "underlying order" in an image involves looking for local self-similarity, that is, looking for portions of the image that either look like or can be made to look like other portions of the image. To illustrate this process, consider the following FIC algorithm for black-and-white images:

**Step 1** Divide the image into $8 \times 8$ pixel blocks called domain blocks. Assume that the length and width of the image are each a multiple 8 pixels.\(^\text{10}\)

**Step 2** Divide the image into $4 \times 4$ pixel blocks called range blocks.

**Step 3** For each of the range blocks, find a matching domain block. The matching domain block is the one that best fits the equation $T(d) = r$, where $d$ and $r$ are the domain and range block respectively and $T$ is a local contractive map. To simplify the search process, $T$ is restricted to maps of the form $s \cdot A$, where $s$ is a scaling factor.

---

\(^{10}\) The image can be padded with “whitespace” (blackspace in this case) if the length or width is not a multiple of 8 pixels. The prospect of compressing a bigger image is counterbalanced by the fact that the whitespace adds self-similarity to the image.
of either 0 or 1 and $A$ is an affine transformation that scales by $\frac{1}{2}$ in the $x$ and $y$
directions and then reorients the square in one of the 8 possible ways of matching
up corners.

The value of $s$ from Step 3 is largely cosmetic, since a value of 1 has no effect. An $s$-value
of 0 is used either when no pixels are turned on in the range block in question or when turning off
all of the pixels in a range block is a closer approximation than scaling and reorienting any of the
available domain blocks.

The black-and-white algorithm above provides an outline for any FIC algorithm. Once the
candidates for range and domain blocks have been identified, range and domain blocks are matched
up by finding local contractive maps that make the domain block look like the range block. Range
and domain blocks do not have to be squares; a "block" might be an arbitrary collections of pixels.
In addition, the same collection of domain blocks does not have to be used for each range block; the
domain blocks depend upon the range blocks.

There are a couple of restrictions on the choices of domain and range blocks. First, the domain
blocks for a range block should be bigger than the range block. Local contractive maps are required
for FIC to work. Second, the range blocks should partition the image (Fisher 910), that is:

- Every pixel should be included in at least one range block. If a pixel is not part of any range
  block, the color(s) for that portion of the image can’t be reproduced.
- No pixel should be included in more than one range block. If a pixel is part of two range blocks,
  then it is possible to produce conflicting color values for that part of the image.

More explanation is required to make the transition from the black-and-white FIC algorithm
to an algorithm that will work on grayscale images. A grayscale algorithm is important not only to
grayscale images but also to color images, since a color algorithm can be built upon 3 copies of the
grayscale algorithm.

As mentioned before a grayscale image can be represented as a function $f: [0,1]^2 \rightarrow [0,1]$. The Hausdorff distance function described before will not work for comparing the distance between
two image functions. In a sense, $f$ occupies all of $[0,1]^2$, since it has function values throughout.
Therefore, a distance function for grayscale images must consider where the images vary, in their color values.

The shift in focus for the distance function also suggests a shift in focus for the local contractive maps. Contraction is now in terms of the grayscale values. Thus, the fixed point of a grayscale local IFS, will be another image function. Range and domain blocks are picked as in the general description of an FIC algorithm. The local contractive maps for a domain/range block pair take the form $s \cdot A(D) + o$, where $A$ reshapes the domain block to look like the range block\(^{11}\), $s$ is a grayscale scaling value that is applied to all of the grayscale values from $A$, and $o$ is a grayscale translation value that is applied after the scaling.

Finding the best $s$- and $o$-values for a range/domain block pair is much more complicated than picking an $s$-value for a black-and-white local contractive map. This process is simplified greatly by considering the pixels involved. Remember, a range or domain block is just a selection of pixels from the image. The pixels from the range block can be ordered and labeled $a_1, a_2, \ldots, a_n$. Likewise, after applying $A$ to the domain block's pixels, the resulting pixels can be ordered and labeled $b_1, b_2, \ldots, b_n$, where $b_i$ maps to the same location as $a_i$. Finding the best $s$- and $o$-values means satisfying the equations:

$$a_i = s \cdot b_i + o, \quad \forall \ i \in \{1, 2, \ldots, n\}$$

or minimizing the root mean squared error (Fisher 913):

$$\text{rms} = \sum_{i=1}^{n} (s \cdot b_i + o - a_i)^2$$

Solving the equations $\frac{\partial \text{rms}}{\partial s} = 0$ and $\frac{\partial \text{rms}}{\partial o} = 0$ provides the equations for $s$ and $o$ that minimize $\text{rms}$:

$$s = \frac{n \sum_{i=1}^{n} (b_i a_i) - \sum_{i=1}^{n} b_i \sum_{i=1}^{n} a_i}{n \sum_{i=1}^{n} b_i^2 - (\sum_{i=1}^{n} b_i)^2}$$

$$o = \frac{\sum_{i=1}^{n} a_i - s \sum_{i=1}^{n} b_i}{n}$$

The $\text{rms}$-value for a domain/range block pair serves as a measure of the distance between the blocks. An $\text{rms}$-value of 0 means that the domain block could be transformed into an exact

\(^{11}\) In terms of actual implementation, $A$ associates clumps of pixels from the domain block with pixel locations in the range block.
duplicate of the range block for some $s$- and $o$-values. Otherwise, the magnitude of the $rms$-value depends upon the number of pixels at which the transformed domain block and range blocks differ and the amount of difference in grayscale values. Similarly, the $rms$-value serves as a measure of the compression loss. When compressing with FIC, the user specifies the maximum acceptable loss, that is, the maximum acceptable $rms$-value for a local contractive map.

Given the shift in contractivity toward the grayscale values, it appears that the requirement that domain blocks be bigger than range blocks is no longer necessary. Although the contractivity (with respect to the Euclidean metric) of the $A$ part of the local IFSes does not contribute to the contractivity of the local IFS, it is still important to FIC. For example, if the domain and range blocks are both the $8 \times 8$ pixel blocks that cover the image, then many images will be compressed with noticeable loss:

Let $r_i$ be one of the range blocks of the image. Let $d_i$ be the domain block chosen for $r_i$. Note: $d_i = r_j$ for some range block $r_j$, since the range and domain blocks are chosen from a common pool. Starting with an image that is a uniform shade of 0.5 and repeatedly applying the local IFS should produce a close approximation of the original image.

After the first application of the local IFS, each range block’s grayscale values have been scaled and translated uniformly within each range block. Thus, the image now consists of $8 \times 8$ squares of possibly different shades. Thus $d_i$ is uniform in shade. Therefore, after the second application of the local IFS, the grayscale values in $r_i$ will be shifted and translated, but they will remain a uniform shade, as will all of the other range blocks. Hence, repeated applications of the local IFS will produce an image that consists of $8 \times 8$-pixel blobs of color.

The problem in the previous example is that the domain blocks did not touch multiple range blocks. It is possible to touch multiple range blocks with a domain block that is smaller than the range block it represents. However, smaller domain blocks will result in a local IFS that does not have a fixed point:
Suppose that $f$ is the fixed point of the local IFS whose domain blocks are smaller than its range blocks. Thus applying the local IFS to $f$ should produce $f$. Applying the local IFS to $f$ will tend to make $f$ more “chunky,” since each of the local contractive maps blow up a domain block. Therefore, $f$ can’t be the fixed point of the local IFS.

Using domain blocks that touch multiple range blocks and are the same size as their range blocks has some promise, but it tends to go against the “feel” of a local IFS. Each application of the local IFS is supposed to be a refinement of detail, which suggests shrinking the domain blocks.

Partitioning has been mentioned throughout this section, but little has been said about how a partitioning scheme should work. The Quadtree partitioning scheme demonstrates the major elements that may be present in a partitioning scheme. After discussing Quadtree partitioning, the overview of FIC will end with a generalized description of the grayscale FIC algorithm.

The Quadtree partitioning algorithm assumes that the length and width of the image (in pixels) are both a power of 2. It also assumes a user-specified value $r$, which is the maximum allowable RMS error. The starting range blocks are the 4 quadrant rectangles that make up the image. For these range blocks, the only possible domain block is the image itself. Given a range block, each domain block is tried until applying one of the 8 symmetries produces an RMS error less than $r$ or the list of domain blocks is exhausted. If a matching domain block is found, the local contraction map is recorded and the range block is discarded from the list of candidates.

If no matching domain block is found, then the range block is divided into its 4 constituent quadrants, and they are added to the list of range blocks. The domain blocks for the new range blocks are all blocks bigger than the range blocks. For example, the second level of range blocks could use the image and the 4 quadrants of the image as range blocks. This process of division continues until either a match is found or a minimum size is reached. If the minimum size is reached, then the best domain block from the last batch is used. The partitioning finishes when the range list is empty.

The Quadtree algorithm can be generalized to produce a grayscale partitioning algorithm. An
object-oriented implementation of this algorithm will be described later in the paper. The following is a verbal description of the grayscale partitioning algorithm:

**Step 1** Generate the initial list of range blocks.

**Step 2** Remove the current range block out the list of range blocks. Call it \( r \). If the list of range blocks is empty, then quit compressing.

**Step 3** Generate the list of domain blocks for \( r \). Set \( t \) to \( \infty \).

**Step 4** Remove the current domain block out of the list of domain blocks. Call it \( d \). If the list of domain blocks is empty, go to **Step 7**.

**Step 5** Pick the best (least RMS error) local contractive map from \( d \) to \( r \). If the RMS error is less than the tolerance value, then record this map and go to **Step 2**.

**Step 6** If the RMS error from **Step 5** is smaller than \( t \), then set \( t \) to the RMS error from 5 and save the map from 5.

**Step 7** If \( r \) is indivisible, go to **Step 8**. Otherwise, divide \( r \) into sub-range blocks and add them to the range block list. Where the new range blocks should be added to the list depends upon the partitioning scheme. Go to **Step 2**.

**Step 8** Record the last saved map from **Step 6** and go to **Step 2**.
CHAPTER 4

FIC IMPLEMENTATION

As mentioned before, this section describes an object-oriented implementation of Fractal Image Compression. An understanding of the following terms related to object-oriented programming is assumed in the description below:

class— A logical grouping of information and ways of manipulating that information. Classes are usually based on "things," not actions. For example, a set class would contain information describing the elements of the set and would provide the means to check whether an element was in the set, complement the set, union two sets, etc. Classes are usually represented as record or structure types that also include routine definitions as fields.

object— An instance of a class. The relationship between classes and objects is analogous to the relationship between types and variables.

method— One of the "ways of manipulating that information" provided by a class. A method is a routine declared within a class. It has access to the data fields of its class. A common practice in object-oriented programming is to prevent direct access to a class's data fields, requiring the programmer to provide methods to do whatever is needed. A class usually includes a couple of special methods called a constructor and a destructor. A constructor initializes the data fields of its object to the values expected by the methods; it is called before any other method of the object. A destructor performs any clean-up work, such as deallocating memory; it is called once the program is finished with an object.
inheritance— Refers to the ability to base a class on a previously-defined class. The descendant class "inherits" all of its ancestor's data fields and methods; new data fields and methods can be added; also, existing methods can be overridden. Inheritance is one of the major features of object-oriented programming. It allows the programmer to customize previous work to fit a particular situation.

polymorphism— Related to inheritance and overriding methods. Polymorphism can be explained with an example. Consider a triangle class with a descendant square class. Each of these classes has its own Area method which returns the area. A pointer to an class will also serve as a pointer to any of its descendants, so it is possible to construct a linked list of triangles and squares. As a result of polymorphism, calling the Area method for a particular list node will call the correct Area method, even though the list nodes are pointers to triangle objects. This assumes that the Area method is declared as a virtual method. Declaring a method as virtual requests that a polymorphic method lookup be done in a case like the linked-list example. If the Area method was not declared as a virtual method, the Area method for the triangle class would be called for every node in the list.

This implementation of FIC was originally written in object-oriented Turbo Pascal. It was later ported to C++. In the class descriptions that follow, the C++ source code will be used. A full listing of the source code (both versions) can be found in the Appendix.

The previous sections have frequently mentioned a "matrix of pixel values;" therefore, it makes sense to implement a "matrix of pixel values" class. The TMemScreen class is an arbitrary matrix of grayscale pixel values. Each pixel is described by a one-byte integer (for grayscale intensities from 0...255). The TMemScreen class contains the following data fields:

```c
int width, height;
unsigned char * matrix;
```

The matrix field is actually a vector of pixel values. This approach was taken for efficiency. The alternative, allocating a list of row pointers and then allocating each row, would involve dereferencing two pointers for each matrix lookup. The vector approach requires some extra math for each lookup,
but the extra math should be less overhead than the extra pointer reference.

All of the fields above are declared in the private: section of the class, that is, they can be referenced only by matrix methods. These fields are initialized by the class’s constructor:

```cpp
TMemScreen(int wid, int ht) {
    width = wid; // Save dimension information
    height = ht;

    matrix = new char[wid*ht]; // Allocate the matrix
}
```

The matrix of pixels can be manipulated with the set and get methods. These methods are declared inline for efficiency; not only does inlining avoid the overhead of a function call for each matrix reference, but it also allows the operations that make up a matrix reference to be optimized for the particular situation, if an optimizing compiler is being used.

```cpp
void set(int x, int y, unsigned char value) {
    matrix[y*width+x] = value;
}
unsigned char get(int x, int y) {
    return matrix[y*width+x];
}
```

Once a particular TMemScreen object is deallocated, its destructor will be called. The destructor is in charge of cleaning up, that is, deallocating the memory allocated by the constructor:

```cpp
~TMemScreen() {
    delete [] matrix;  // Deallocate the matrix
}
```

The TMemScreen class is used throughout this object-oriented implementation of FIC. However, unlike most of the other classes, TMemScreen has no descendants. It is a simple class that is complex enough to model a “matrix of pixel values.”

The idea of a block of pixels pops up repeatedly in the description of the FIC algorithm, so it is another good candidate for a class. The TBlock class is the ancestor of all of the real “block of pixels” classes. It describes the general form of “block of pixels” class. It contains no data fields, since there are no data fields that would appear in rectangular, triangular, arbitrary, etc. block-of-pixels types. Similarly, the methods do nothing beyond cataloging the methods that must be provided by any

---

12 In addition to public: and private: sections, a class can also have a protected: section. The fields and methods in the protected: section of a class are accessible by the class and its descendants.
block-of-pixels type:

```cpp
TBlock() {}
virtual ~TBlock() {};

virtual Display() {};
virtual TFloat MinimizeRMS(TMemScreen * Screen, TBlock * Domain,
    TTransform * Transform)
{};
```

Display is a debugging output routine. MinimizeRMS looks at the possible ways of mapping Domain to this block and returns the result in Transform. The RMS error is returned by the function. TBlock is an abstract class. Its main purpose is to provide a pointer type that can be used to reference any of its more concrete descendants. A new “type” of block can be defined as follows:

1) Define a descendant class of TBlock which describes the block properties that are common to range and domain blocks of this type. For example, one could define a TSquareBlock class containing location and size information.

2) Define range and domain descendant classes of the class from 1). In the example from 1), one would need to define TSquareDomainBlock and TSquareRangeBlock classes.

The main partitioning scheme demonstrated by this implementation of FIC is the quadtree algorithm, which uses rectangular blocks. The following class descriptions demonstrate what is involved in defining a new block organization.

TRectangularBlock describes a rectangular chunk of pixels from an image. The actual pixel values are not stored within this class, just the location and dimensions of the block; a TMemScreen object is referenced when the actual pixel values are needed. TRectangularBlock is defined as follows:

---

13 The block hierarchy could also be maintained by multiple inheritance. Multiple inheritance would require defining two more abstract classes TRangeBlock and TDomainBlock which describe generic range and domain blocks respectively. In the example above, TSquareBlock would be replaced by TSquare, which encapsulates the “squareness” of TSquareBlock but is not descended from TBlock; TSquareRangeBlock would be derived from TSquare and TRangeBlock; TSquareDomainBlock would be derived from TSquare and TDomainBlock. The single-inheritance approach was chosen to simplify the object hierarchy.

14 Many versions of the quadtree algorithm require the starting block to be a square, thus making all of the subblocks squares. This requirement is not present in the quadtree algorithm described by this paper.
class TRectangularBlock : public TBlock
{
    protected:
    int ulx, uly,
       width,
       height;

    public:
    TRectangularBlock(int x, int y, int wid, int ht) : TBlock()
    { ulx = x; uly = y; width = wid; height = ht; }

    virtual void Display();
    int Get_ulx() { return ulx; } // Read-only access to
    int Get_uly() { return uly; } // protected fields.
    int Get_width() { return width; }
    int Get_height() { return height; }
};

The data fields are declared in the protected section of the class, so they are not directly accessible outside the class (or its descendants). The Get_field methods do provide read-only access to these fields, however. These methods are defined inline, so block.Get_ulx() should incur no more overhead than block.ulx would if ulx was a public field.

In general, the definition of a TShapeBlock should identify the pixels associated with the block (by TMemScreen coordinates) and provide a way to order these pixels. In addition, it should contain any fields or methods that are common to its range and domain blocks.

As mentioned before, the TRectRangeBlock and TRectDomBlock classes need to be derived before anything can be done with rectangular blocks. These classes are responsible for most of the low-level work done in any partitioning scheme based on rectangular blocks. The descriptions of these classes follow:

class TRectRangeBlock : public TRectangularBlock
{
    protected:
    TFloat RangeSum,
       RngSqrSum;

    public:
    TRectRangeBlock(int x, int y, int wid, int ht, TMemScreen * Image);

    virtual void Display();
    virtual TFloat MinimizeRMS(TMemScreen * Image, TBlock * Domain,
                                      TTransform * Trans);
};

class TRectDomBlock : public TRectangularBlock
The data fields in the classes above are used hold preprocessing information. This preprocessing information will be used in MinimizeRMS calls. As long as the RMS formula described before is used as a basis for calculating error values, analogous data fields should appear in the range and domain variants for other blocks shapes. The following table matches RMS-formula terms with data fields:

<table>
<thead>
<tr>
<th>Term</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum_{i=1}^n a_i$</td>
<td>RangeSum</td>
</tr>
<tr>
<td>$\sum_{i=1}^n a_i^2$</td>
<td>RangeSqrSum</td>
</tr>
<tr>
<td>$\sum_{i=1}^n b_i$</td>
<td>DomSumx</td>
</tr>
<tr>
<td>$\sum_{i=1}^n b_i^2$</td>
<td>DomSqrSumx</td>
</tr>
<tr>
<td>$n^2 \sum_{i=1}^n b_i^2 - (\sum_{i=1}^n b_i)^2$</td>
<td>SDenomx</td>
</tr>
</tbody>
</table>

The preprocessing work involved in filling in the range and domain block data fields is done by their respective constructors. It is important to note that a particular domain block is dependant upon a corresponding range block, or more specifically, the dimensions of a corresponding range block. The RMS formulas use the same number of pixel values from the range block as from the domain block, which means either that a subset of the domain-block pixels will be used or multiple domain-block pixels will be combined. TRectDomBlock takes the latter approach; rectangular clumps of pixels are averaged.

Averaged pixel values are stored in the DomXisX and DomXisY fields; DomXisX holds the pixel
values that result from mapping the domain block down to the size of the range block; DomXisY holds the pixel values that result from rotating the domain block by 90 degrees and then mapping it down to the size of the range block. All of the xxx1 fields are generated from the DomXisX pixel values. Likewise, all of the xxx2 fields are generated from the DomXisY pixel values.

Before discussing MinimizeRMS, the most important method from the classes above, some background needs to be covered about the TTransform class and its descendants. A transform is a way of mapping from a domain block to a range block, that is, it is one of the local contractive maps of the local IFS being built. TTransform is defined as follows:

```cpp
class TTransform
{
public:
    TBlock *Range, *Domain;
    TFloat Grayscale, GrayTrans;

    TTransform(TBlock * R, TBlock * D, TFloat S, TFloat O)
    { Range = R; Domain = D; Grayscale = S; GrayTrans = O; }

    virtual ~TTransform() {};

    virtual void DisplayO;
};
```

TTransform contains no methods to do any real manipulation of its data. It is treated more like a structure type than a class. That is why its data is declared in the public section. The class does not depend upon data consistency, so it is not necessary to hide the data beneath a veil of data-accessing methods. As their names suggest, Range and Domain are pointers to the range and domain blocks respectively. Grayscale and GrayTrans are the s and o values calculated by the RMS formulas.

Before a block class to be used within a partitioning algorithm, an associated transform class has to be derived. Associated with the TRectangularBlock class is the TRectTransform class. TRectTransform introduces a Symmetry field that tells which of the 8 ways of matching up the corners of the range and domain blocks is being used. The TSymmetry type, which contains the 8 symmetries, is defined as follows:

```cpp
enum TSymmetry {identity, reflect_y, reflect_x, rot180, reflect_y_eq_x, rot90, rot270, reflect_y_eq_minus_X};
```
The **MinimizeRMS** method of a range block fills in a transform object of the same “type” as the range block. Filling in a transform object means setting the **Range** and **Domain** pointers, calculating \( s \) and \( o \) and storing them in **GrayScale** and **GrayTrans** respectively, and filling in any fields that are specific to this class of transform, for example, the symmetry in the case of a rectangular transform.

The **MinimizeRMS** method for **TBlock** should not be called, since **TBlock** does not know which pixels make up the block—meaning that each new **TshapeRangeBlock** class must override the **MinimizeRMS** method. The **MinimizeRMS** method for **TRectRangeBlock** calculates the RMS values for each of the eight symmetries and returns a transform with the smallest RMS value. The full source code for this method appears in the Appendix. However, the following selections from the method demonstrate how it works:

```c
for (Sym = identity; Sym <= reflect_y_eq_minus_x; Sym++) {
    RngDomSum = 0;
    if (Sym < reflect_y_eq_x) {
        DomGrays = ((TRectDomBlock *)Domain)->DomXisX;
        DomSum = ((TRectDomBlock *)Domain)->DomSum1;
        DomSqrSum = ((TRectDomBlock *)Domain)->DomSqrSum1;
        SDenom = ((TRectDomBlock *)Domain)->SDenom1;
    } else {
        ...
    }
}
switch (Sym) {
    case identity:
    case reflect_y_eq_x:
    case reflect_x:
    case rot90:
        XDir = 1;
        DomStartCol = 0;
        break;
    case rot270:
        YDir = 1;
        DomRow = 0;
        break;
    default:
        XDir = -1;
        DomStartCol = width-1;
        break;
}
```
MinimizeRMS first calculates $\sum_{i=1}^{n} (b_i a_i)$ for the current symmetry and stores its value in RngDomSum. RngDomSum is used along with the preprocessed information stored in the range and domain blocks to calculate $s$ and $o$ and from them, the RMS error for the symmetry.

The FIC algorithm described in this paper works by partitioning an image into range blocks. Thus, the class that embodies the heart of the FIC process is called a partitioner. As with the block class hierarchy, an abstract class, TPartitioner, is the ancestor of any “partitioner” class. TPartitioner describes the basic structure of the FIC algorithm. It is declared as follows:

```cpp
class TPartitioner {
protected:
  TMemScreen * ImagePtr;
  TFloat tolerance;
  TTransform * CurTrans,
              * MinTrans;

public:
  virtual TBlock * NextRangeBlock() { return NULL; }
  virtual void SetUpDomainBlocks(TBlock * Range) {};
  virtual TBlock * NextDomainBlock() { return NULL; }
  virtual int Encode(TTransform * trans, void ** buffer) { return 0; }
```
virtual int IsDivisible(TBlock * Range) { return 0; }
virtual void Split(TBlock * Range) {};

public:
TPartitioner(TMemScreen * image, TFloat t)
{ ImagePtr = image;
  tolerance = t;
}
virtual ~TPartitioner() {};

virtual int Get_width() { return ImagePtr->Get_width(); }
virtual int Get_height() { return ImagePtr->Get_height(); }
virtual int Get_type() { return 0; }
virtual int NextTransform(void ** BufPtr);
};

A "partitioner" object is initialized with a TMemScreen object describing the image to compress and a real number that represents the acceptable level of error in transforms chosen. Once a "partitioner" object has been initialized, compressed image data is returned by repeated calls to NextTransform. NextTransform is declared as follows:

int TPartitioner::NextTransform(void ** BufPtr)
{
  TBlock * Range,
  * Domain;
  TTransform * TempTrans;
  TFloat RMS,
  MinRMS = NearlyInfinite;
  int FoundDom = 0,
  len;

  Range = NextRangeBlock();
  if (Range == NULL) return(0); // Out of range blocks

  SetUpDomainBlocks(Range);
  while (Domain = NextDomainBlock())
  {
    RMS = Range->MinimizeRMS(ImagePtr, Domain, CurTrans);
    if (RMS < MinRMS)
    {
      MinRMS = RMS;
      TempTrans = MinTrans; // Swap pointers
      MinTrans = CurTrans;
      CurTrans = MinTrans;

      FoundDom = (RMS<=tolerance);
      if (FoundDom) break; // Found a good enough match
    }
  }

  if (FoundDom || !IsDivisible(Range))
  {
    #ifdef VERBOSE
    MinTrans->Display(); // Debugging output
    #endif
  }
}
printf("RMS error = %f\n", RMS);
#endif
len = Encode(MinTrans, BufPtr); // Return the best match found
}
else
{
    Split(Range); // Divide into sub-range blocks
    len = NextTransform(BufPtr);
}

return len;

NextTransform returns a pointer to a buffer containing encoded transform data and the buffer length. A buffer length of 0 is returned when the entire image has been partitioned. The NextTransform method used by TPartitioner is suitable for use with any partitioning scheme. Descendant "partitioner" objects need to override the private methods used by NextTransform.

NextRangeBlock returns a pointer to the range block to be considered for transformation. Whether the range block is removed from a list or calculated based on the current context of the partitioning process depends upon the type of partitioning being done. SetUpDomainBlocks prepares for subsequent NextDomainBlock calls. SetUpDomainBlocks takes a pointer to the current range block as a parameter since the pool of available domain blocks may depend upon the range block being considered. NextDomainBlock returns a pointer to the next domain block for a particular range block.

Encode converts a transform into a sequence of bytes that uniquely describe that transform. This sequence of bytes will be handed off to an entropy encoder, which will compress the transform data and eventually write it to a file. IsDivisible returns a non-zero value if the range block passed in can be subdivided into more range blocks. Split is called to divide a range block into sub-range blocks. In fact, Split is not limited to subdividing the range block in question; a call to Split could rearrange all of the unused range blocks.

The quad-tree-partitioning class, TQuadPart demonstrates how to derive a descendant class of TPartitioner. TQuadPart implements the quad-tree algorithm described in the previous section.

---

15 NextTransform returns information about one transform per call, but it could be overridden to return information about more than one transform per call, i.e., efficiency could be improved by decreasing the number of NextTransform calls.
It is declared as follows:

class TQuadPart : public TPartitioner
{
protected:
  TRectDomBlock * CurDom;
  TSLL RangeList,
       DomLists;
  int MaxDomWid,
       MaxDomHt;

  virtual TBlock * NextRangeBlock();
  virtual void SetUpDomainBlocks(TBlock * Range);
  virtual TBlock * NextDomainBlock();
  virtual int Encode(TTransform * trans, void ** buffer);
  virtual int IsDivisible(TBlock * Range);
  virtual void Split(TBlock * Range);

  virtual void SetUpDomainLists();
  virtual void BuildDomainLists(TBlock * Domain);
  virtual void SetUpRangeBlocks(TBlock * Domain);

public:
  TQuadPart(TMemScreen * image, TFloat t) : TPartitioner(image, t)
  {
    TRectangularBlock * TempDom;
    MinTrans = new TRectTransform(NULL, NULL, 0, 0, identity);
    CurTrans = new TRectTransform(NULL, NULL, 0, 0, identity);
    SetUpDomainLists();
    TempDom = new TRectangularBlock(0,0,image->Get_width(),
                                     image->Get_height());
    SetUpRangeBlocks(TempDom);
  }
  virtual ~TQuadPart()
  {
    delete MinTrans;
    delete CurTrans;
  }

  virtual int Get_type() { return QuadTreePartition; }
  virtual void Display() { RangeList.Display(); DomLists.Display(); }
};

The private section introduces a class called TSLL. TSLL is a singly-linked list class. TSLL works on TSLLItem objects or descendants of TSLLItem. For more information, see the Appendix. The list of range blocks to check is maintained within RangeList. The entries of RangeList are a descendant class of TRectRangeBlock called TQuadRectRangeBlock. The only modification is the addition of an integer field called level, which corresponds to the level within the quad-tree. DomLists is a list of all of the possible domain lists. The entries of DomLists are TSLLLItem objects, which are descendants of TSLLItem that have a TSLL field called SubList. DomLists is set up in the interest
of moving as much as possible to a pre-processing stage. CurDom is a pointer to the current domain block within the current domain list being considered.

The final pair of fields, MaxDomWid and MaxDomHt, hold the maximum dimensions of the domain blocks used by the partitioner. The general description of the quad-tree algorithm starts with the entire image as the first domain block. However, the chances of, for example, a 128 x 128 pixel domain block meeting the tolerance requirements are very slim. Thus, for efficiency, TQuadPart makes several divisions before considering domain block.

The 3 new private functions: SetUpDomainLists, BuildDomainLists, and SetUpRangeBlocks are called in the process of TQuadPart's construction:

```c++
void TQuadPart::SetUpDomainLists()
{
    TRectangularBlock * Domain;
    MaxDomWid = ImagePtr->Get_width();
    MaxDomHt = ImagePtr->Get_height();

    while (MaxDomWid*MaxDomHt > MAX_QUAD_PIXELS)
    {
        MaxDomWid = MaxDomWid>>1;
        MaxDomHt = MaxDomHt>>1;
    }

    Domain = new TRectangularBlock(0,0,MaxDomWid,MaxDomHt);
    BuildDomainLists(Domain);
}

void TQuadPart::SetUpRangeBlocks(TBlock * Domain)
{
    register TRectangularBlock * DomPtr = (TRectangularBlock *)Domain;
    int NewWid, NewHt;

    NewWid = DomPtr->Get_width() >> 1;
    NewHt = DomPtr->Get_height() >> 1;

    if (DomPtr->Get_width() == MaxDomWid)
    {
        RangeList.Insert(new TQuadRectRangeBlock(
            DomPtr->Get_ulx()+NewWid,DomPtr->Get_uly()+NewHt,
            NewWid,NewHt,ImagePtr,1));
        RangeList.Insert(new TQuadRectRangeBlock(
            DomPtr->Get_ulx(),DomPtr->Get_uly()+NewHt,
            NewWid,NewHt,ImagePtr,1));
        RangeList.Insert(new TQuadRectRangeBlock(
            DomPtr->Get_ulx()+NewWid,DomPtr->Get_uly(),
            NewWid,NewHt,ImagePtr,1));
        RangeList.Insert(new TQuadRectRangeBlock(
            DomPtr->Get_ulx(),DomPtr->Get_uly(),
            NewWid,NewHt,ImagePtr,1));
    }
```
else
{
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx()+MewWid, DomPtr->Get_uly()+MewHt,
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx(), DomPtr->Get_uly()+MewHt,
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx()+NewWid, DomPtr->Get_uly(),
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx(), DomPtr->Get_uly(),
        NewWid, NewHt));
}
delete Domain;

SetUpDomainLists first calculates MaxDomWid and MaxDomHt and then calls BuildDomainLists to recursively generate all possible domain blocks for this image. SetUpRangeBlocks generates all of the range blocks that are one size under the maximum domain block size. These are the “level 1” range blocks. They are associated with the first domain list, that is, the partitioner will attempt to map domain blocks in the first list to these range blocks. As these range blocks are split, they will generate “level 2,” “level 3,” etc., range blocks.

An analogue to SetUpRangeBlocks is required in any partitioner class. SetUpRangeBlocks represents the preparatory work necessary for NextRangeBlock and Split to work properly. An analogue to SetUpDomainLists, however, is not necessary. Generating all possible domain blocks greatly simplifies the writing of SetUpDomainBlocks and NextDomainBlock and can eliminate redundant domain-block calculations, but it does so at the cost of memory. The main reason that this approach is feasible within TQuadPart is a limitation on the possible range-domain block pairs. Range blocks are only paired with domain blocks that are one level up in the quad-tree. Without this limitation, it would be necessary to generate a list of domain block lists for each size of range block, that is, all of the domain blocks that are bigger than the range block.

In any partitioner class, the workings of NextRangeBlock and Split are closely related. One dispenses the range blocks that make up the partition of the image, and the other restructures the partition to account for a range block that cannot be adequately “approximated.” In TQuadPart,
NextRangeBlock simply removes and returns the head of RangeList. SetUpRangeBlocks adds range blocks in the proper order; that order is maintained as Split adds new range blocks:

```cpp
void TQuadPart::Split(TBlock * Range)
{
    int CurrLevel,
        RngWid, RngHt;
    register TQuadRectRangeBlock * RangePtr = (TQuadRectRangeBlock *)Range;
    RngWid = RangePtr->Get_width()>>1;
    RngHt = RangePtr->Get_height()>>1;
    CurrLevel = RangePtr->Get_level()+1;
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx()+RngWid,RangePtr->Get_uly()+RngHt,
        RngWid,RngHt,ImagePtr,CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx(),RangePtr->Get_uly()+RngHt,
        RngWid,RngHt,ImagePtr,CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx()+RngWid,RangePtr->Get_uly(),
        RngWid,RngHt,ImagePtr,CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx(),RangePtr->Get_uly(),
        RngWid,RngHt,ImagePtr,CurrLevel));
    delete Range; // Finished with this range block
}
```

Split takes a range block and adds the four range blocks that appear below it in the quad-tree. Inserting the new range blocks at the beginning of RangeList guarantees that the sequence of range blocks chosen will be the leaf nodes of the final quad-tree from left to right. The advantage of generating the range blocks in that order is that it reduces the amount of information necessary to encode a transform. TQuadPart transforms are encoded into a structure called TQuadPartRec:

```cpp
struct TQuadPartRec
{
    unsigned DomX : 9;
    unsigned DomY : 9;
    unsigned RngWid : 8;
    unsigned CSignBit : 1;
    unsigned sym : 3;
    unsigned fill : 2;
    short S;
    unsigned char 0;
};
```

DomX and DomY hold the location of the domain block. RngWid is the width of the range block. From RngWid, the decompressor can infer the height and location of the range block and the size
of the domain block. This inference is accomplished by building the quad-tree as transforms are
read. sym, S, and 0 are the familiar symmetry, scale, and translation values from TRectRange-
Block::MinimizeRMS. OSignBit is set to 0 for a positive translation value, 1 for a negative transla-
tion value.

Encode allocates a TQuadPartRec, fills it in, and returns a pointer to the allocated structure.
In general, using some sort of structure to represent an encoded transform is very useful. It avoids
some of the conversion work that would be required to encode into an array of bytes. However, it is
good to be aware of the actual representation that a structure entails, since it is being used to store
compressed data.

In the general case, SetUpDomainBlocks prepares the partitioner for calls to NextDomainBlock.
In some cases, this may mean actually allocating and initializing a list of domain blocks. If a list
of domain blocks is not being maintained in memory, then domain blocks will need to be allocated
and initialized within NextDomainBlock. In TQuadPart, SetUpDomainBlocks merely points CurDom
to the head of the domain list associated with this range block, or, more accurately, associated with
this range block's level. NextDomainBlock traverses the domain list, returning CurDom, until the
end of the list is reached.

IsDivisible provides the cutoff condition for the recursive process of matching and splitting
range blocks. It should return a false value when it is not possible to Split the current range block.
It may also return a false value when the partitioner determines that it is more beneficial to accept
the error in the current transform than to accept the extra storage required for the transforms
that would result from splitting the current range block, for example, when the space required to
represent a transform exceeds the space required to represent the actual pixel values of the range
block. In TQuadPart, a range block is divisible as long as its width and height are both even.

All that remains is to describe input and output of FIC, that is, to describe the first and last
arrows of the following diagram:
Once the `TMemScreen` object is filled in, it will be passed into the constructor of the partitioner. `TInputFile` is the abstract base class for source image files:

```cpp
class TInputFile
{
  protected:
  TMemScreen * ImagePtr;

  public:
  TInputFile() { ImagePtr = NULL; }
  virtual ~TInputFile() { delete ImagePtr; }
  virtual TMemScreen * Get_ImagePtr() { return ImagePtr; }
  virtual int Get_width() { return 0; }
  virtual int Get_height() { return 0; }
};
```

The constructor of a descendant class should open the file and read enough to determine the dimensions of the image, allocate space for the `TMemScreen` and initialize it with the dimension information, and copy the pixel values from the file into the `TMemScreen`. The destructor of a descendant class should close the file. The Appendix contains `TTgaFile`, a sample descendant of `TInputFile` which handles grayscale Targa formats.

¶ 2 is implemented by the `TEncoder` class. Unlike most of the other base classes, `TEncoder` objects are allowed. `TEncoder` represents the simplest type of entropy encoding, that is, the information given to `TEncoder` is written directly to the output file without any compression. `TEncoder` is declared as follows:
class TEncoder {
protected:
    int EncodeType;
    FILE * fp;

    virtual void WriteHeader(TPartitioner * Part);
    virtual void WriteToFile(int len, void * buffer);
    virtual void CloseOutputFile();

public:
    TEncoder(char * name) {
        if ((fp = fopen(name,"w")) == NULL) {
            perror("Error opening the output file");
            exit(1);
        }
        EncodeType = IdentityEncode;
    }

    virtual ~TEncoder() { CloseOutputFile(); }

    virtual void EncodeFIF(TPartitioner * Part);
};

As mentioned before, the encoder communicates with the partitioner by repeatedly calling Next-Transform. Each NextTransform call returns the uncompressed version of a transform chosen by the partitioner. The entropy-encoding scheme can be changed by overriding the WriteToFile method. Depending upon the type of encoding used, CloseOutputFile may also need to be overridden to flush any remaining data before closing the file.

It is important to note that the 3 parts that make up the FIC algorithm: reading the input file, partitioning the image, and compressing the transform data are mutually independent. The partitioner does not care whether its matrix of pixels came from a Targa file, GIF file, random number generator, etc. Likewise, the encoder does not care what type of partitioning scheme is being used to produce the transforms it is compressing.
CHAPTER 5

CONCLUSIONS

Throughout the description of the object-oriented implementation of FIC, care has been taken to point out how each object could be customized. For example, TPartitioner can be overridden to produce a triangular partitioner. The inherent flexibility of an object-oriented language allows for the creation of an infinite number of Fractal Image Compressors.

The breakdown of the algorithm into objects also allows for easy maintenance. Whenever a new object class is derived, it can be tested independently before using it in the FIC algorithm. For example, when a TGIFFile object class is derived from TInputFile, its behavior can be tested without testing the entire FIC algorithm, by testing whether the correct pixel values are read into a TMemScreen object.

Finally, the object-orient approach provides a good description of the FIC algorithm. Each step of the process is represented by an object or method:

![Diagram of FIC Object/Method Breakdown](image)

Figure 4: An FIC Object/Method Breakdown
Examining NextTransform, one sees the interplay between range and domain blocks and sees how the range blocks are partitioned by Split and IsDivisible. Hence, what the algorithm does can be explained without recourse to the mathematics that explain how the algorithm works.
APPENDIX

This section contains the source code of the programs used to test this approach to the FIC algorithm. The C++ source code is listed first. It is followed by the original Turbo Pascal source, upon which the C++ source was based.

C++ Source

GLOBAL.H

#ifndef GLOBAL
#define GLOBAL 1
//
// global.h
//

#include <math.h>

//
// For protecting against roundoff and overflow
//
#define NearlyZero .5E-10
#define NearlyInfinite .5E10
#endif // GLOBAL

SLLIST.H

#ifndef SLLIST
#define SLLIST 1
//
// sllist.h
//
// General-purpose object-oriented singly linked lists. This is
// a bare-bones implementation, using only a head pointer for
// the singly linked list. There is no tail pointer and no size
// field.
//
// Also, the TSLCL circularly-linked list is not really circularly
// linked at all - it uses the same internal storage form as
// TSLL. The only difference is that the Next function is
// overridden to report the head of the list instead of nil at
// the end of the list and the Prev function is similarly
// overridden to behave in circularly linked list fashion at the
// head of the list. The Display procedure is also overridden so
// it will not loop forever.
//
#include <stddef.h>

//
// TSLLItem is an abstract data object, so that you should never
// create an instance of it. It merely serves as the common ancestor
// for all those concrete objects which may appear on a singly
// linked list.
//
class TSLLItem
{
  public:
    TSLLItem *Next; // The only data item here is a pointer to the
    // next item on the list.

    TSLLItem() {Next = NULL;} // Unconnected item
  virtual ~TSLLItem() {}

  virtual void Display() {} // For debugging ancestor objects
};

// A singly-linked list of TSLLItem's.
//
class TSLL
{
  public:
    TSLLItem *HeadPtr;

    TSLL() {HeadPtr=NULL;} // Initialize to an empty list
  virtual ~TSLL(); // Destroys the list (and all the items on it).

  void Insert(TSLLItem *PItem);
    // Inserts an element at the start of the list.
  void Append(TSLLItem *PItem);
    // Appends an element at the end of the list.
  void PlaceAfter(TSLLItem *NewItem, TSLLItem *OldItem);
    // Here OldItem points to an item which is already
    // somewhere on the list. PlaceAfter placesNewItem
    // on the list following OldItem.
  void PlaceBefore(TSLLItem *NewItem, TSLLItem *OldItem);
    // Here OldItem points to an item which is already
    // somewhere on the list. PlaceBefore placesNewItem
    // on the list before OldItem.
  void Push(TSLLItem *PItem);
    // Inserts an element at the start of the list - just
    // like Insert - for the times when you want the
    // singly linked list to behave like a stack.
  void Delete(TSLLItem *PItem);
    // Deletes an item from the list and destroys the item.
  void Remove(TSLLItem *PItem);
    // Removes an item from the list, but does not
    // destroy the item.

  TSLLItem * Pop(); // Returns a pointer to the (old) head of the list,
    // which is removed from the list for the times
    // when you want to treat the singly linked list as a
    // stack.
  TSLLItem * Head(); // Returns the first item on the list.
  TSLLItem * Tail(); // Returns the last item on the list.
  virtual TSLLItem * Next(TSLLItem *PItem);
    // Here PItem is an item which is already somewhere
    // on the list. Next returns the item following PItem
    // on the list.
  virtual TSLLItem *Prev(TSLLItem *PItem);
// Here PItem is an item which is already somewhere on the list. Prev returns the item preceding PItem on the list.
TSLLItem *Nth(int n);
// Returns the nth item on the list. If n is not the number of an item on the list, Nth returns nil.
int Empty();  // Returns true if the list is empty; false otherwise.
int Size();   // Returns the number of items on the list.
virtual void Display();
// Displays the items on the list in normal order, calling each object's Display routine.

// A singly-linked circular list.

class TSLCL : public TSLL
{
public:
    TSLCL() : TSL() {}

    virtual TSLLItem *Next(TSLLItem *PItem);
        // Here PItem is an item which is already somewhere on the list. Next returns the item following PItem on the list.
    virtual TSLLItem *Prev(TSLLItem *PItem);
        // Here PItem is an item which is already somewhere on the list. Prev returns the item preceding PItem on the list.
    virtual void Display();
        // Displays the items on the list in normal order, calling each object's Display routine.
};

// A singly-linked list entry that contains a singly-linked list.

class TSLLLItem : public TSLLItem
{
public:
    TSL SubList;

    TSLLLItem() : TSLLItem() {}
    virtual ~TSLLLItem() {}

    virtual void Display() { SubList.Display(); }  
        // Displays the sublist's contents
};

#endif // SLLIST

SLLIST.CC

//
// SLLIST.CC
//
#include "sllist.h"

// TSLL Methods
// TSLL::TSLL()
{ 
    while (HeadPtr != NULL)
        Delete(HeadPtr);
}

void TSLL::Insert(TSLLItem *PItem)
{ 
    PItem->Next = HeadPtr;
    HeadPtr = PItem;
}

void TSLL::Append(TSLLItem *PItem)
{ 
    if (HeadPtr == NULL)
        Insert(PItem);
    else
        Tail()->Next = PItem;
}

void TSLL::PlaceAfter(TSLLItem *NewItem, TSLLItem *OldItem)
{ 
    if (OldItem == Tail())
        Append(NewItem);
    else
    { 
       NewItem->Next = OldItem->Next;
        OldItem->Next = NewItem;
    }
}

void TSLL::PlaceBefore(TSLLItem *NewItem, TSLLItem *OldItem)
{ 
    if (OldItem == HeadPtr)
        Insert(NewItem);
    else
    { 
       NewItem->Next = OldItem;
        Prev(OldItem)->Next = NewItem;
    }
}

void TSLL::Push(TSLLItem *PItem)
{ 
    PItem->Next = HeadPtr;
    HeadPtr = PItem;
}

void TSLL::Delete(TSLLItem *PItem)
{ 
    Remove(PItem);
delete(PItem); // Free the memory for this item
}

void TSLL::Remove(TSLLItem *PItem)
{
    if (PItem == HeadPtr)
        HeadPtr = HeadPtr->Next;
    else
        Prev(PItem)->Next = PItem->Next;
PItem->Next = NULL;
}

TSLLItem *TSLL::Pop()
{
    Remove(HeadPtr);
    return(HeadPtr);
}

TSLLItem *TSLL::Head()
{
    return(HeadPtr);
}

TSLLItem *TSLL::Tail()
{
    TSLLItem *p, *q;

    if (HeadPtr == NULL)
        return(NULL);
    else
    {
        q = HeadPtr;
p = HeadPtr->Next;
        while (p != NULL)
        {
            q = p;
p = p->Next;
        }

        return(q);
    }
}

TSLLItem *TSLL::Next(TSLLItem *PItem)
{
    return(PItem->Next);
}

TSLLItem *TSLL::Prev(TSLLItem *PItem)
{
    TSLLItem *p, *q;

    q = NULL;
p = HeadPtr;
    while (p != PItem)
    {
        q = p;
p = p->Next;
    }

    return(q);
}
TSLLItem *TSLL::Nth(int n) {
    TSLLItem *p;
    int count;

    if (n <= 0)
        return(NULL);
    else {
        p = HeadPtr;
        count = 1;
        while (count < n && p != NULL) {
            p = p->Next;
            count++;
        }
        return(p);
    }
}

int TSLL::Empty() {
    return(HeadPtr == NULL);
}

int TSLL::Size() {
    TSLLItem *p;
    int n = 0;
    p = HeadPtr;
    while (p != NULL) {
        n++;
        p = p->Next;
    }
    return(n);
}

void TSLL::Display() {
    TSLLItem *p;

    p = HeadPtr;
    while (p != NULL) {
        p->Display();
        p = p->Next;
    }
}

//
// TSLCL Methods
/
 TSLLItem *TSLCL::Next(TSLLItem *PItem)
{
    if (PItem->Next == NULL)
        return(HeadPtr);
    else
        return(PItem->Next);
}

TSLLItem *TSLCL::Prev(TSLLItem *PItem)
{
    if (PItem == HeadPtr)
        return(Tail());
    else
        return(TSSL::Prev(PItem));
}

void TSLCL::Display()
{
    TSLLItem *p;

    if (HeadPtr != NULL)
    {
        HeadPtr->Display();
        p = HeadPtr->Next;
        while (p != HeadPtr)
        {
            p->Display();
            p = p->Next;
        }
    }
}

BLOCK.H

#ifndef BLOCK
#define BLOCK 1
//
// block.h
//

#include <stdio.h>
#include <stdlib.h>
#include "sllist.h"

// Amount by which to multiply the "S" values when converting to a integer
#define GSMult 512.0

#ifndef USE_SINGLE
typedef float TFloat;
#else
    typedef double TFloat;
#endif

//
// Matrix of pixels

class TMemScreen
{
private:
    int width, height;
    unsigned char * matrix;

public:
    TMemScreen(int wid, int ht)
    {
        width = wid;   // Save dimension information
        height = ht;

        matrix = new unsigned char[wid*ht]; // Allocate the matrix
    }

    ~TMemScreen()
    { delete [] matrix; } // Deallocate the matrix

    void set(int x, int y, unsigned char value)
    { matrix[y*width+x] = value; } // Set a pixel value

    unsigned char get(int x, int y)
    { return matrix[y*width+x]; } // Get a pixel value

    int Get_width()
    { return width; }

    int Get_height()
    { return height; }

    int AvgGray(int ulx, int uly, int wid, int ht);
};

class TTransform; // Forward reference

//
// Generic range/domain block
//
class TBlock : public TSLLItem
{
    public:
        TBlock() : TSLLItem() {}
        virtual ~TBlock() {}

        virtual void Display() {}
        virtual TFloat MinimizeRMS(TMemScreen * Image, class TBlock * Domain, TTransform * Trans) {}
};

class TRectangularBlock : public TBlock
{
    protected:
        int ulx, uly,
            width,
            height;

    public:
        TRectangularBlock(int x, int y, int wid, int ht) : TBlock()
        { ulx = x; uly = y; width = wid; height = ht; }

        virtual void Display();
    int Get_ulx() { return ulx; } // Read-only access to
int Get_uly() { return uly; } // protected fields.
int Get_width() { return width; }
int Get_height() { return height; }

class TRectRangeBlock : public TRectangularBlock
{
protected:
  TFloat RangeSum,
    RngSqrSum;
public:
  TRectRangeBlock(int x, int y, int wid, int ht, TMemScreen * Image);
  virtual void DisplayO;
  virtual TFloat MinimizeRMS(TMemScreen * Image, TBlock * Domain,
    TTransform * Trans);
};

class TRectDomBlock : public TRectangularBlock
{
protected:
  int     RngWid,RngHt;
  TMemScreen *DomXisX,
    *DomXisY;
  TFloat  DomSum1,DomSum2,
    DomSqrSum1,DomSqrSum2,
    SDenom1, SDenom2;
public:
  TRectDomBlock(int x, int y, int wid, int ht, class TBlock * Range,
    TMemScreen * Image);
  virtual ~TRectDomBlock() { delete DomXisX; delete DomXisY; }
  virtual void DisplayO;
  friend TFloat TRectRangeBlock::MinimizeRMS(
    TMemScreen * Image, TBlock * Domain,
    TTransform * Trans);
};

class TQuadRectRangeBlock : public TRectRangeBlock
{
protected:
  int level;
public:
  TQuadRectRangeBlock(int x, int y, int wid, int ht, TMemScreen * Image,
    TTransform * Trans, int l) : TRectRangeBlock(x,y,wid,ht,Image)
  { level = l; }
  virtual int Get_level() { return level; }
  virtual void DisplayO()
  {
    TRectRangeBlock::DisplayO;
    printf("Level = %d\n", level);
  }
};
class TTransform
{
public:
  TBlock *Range,
  *Domain;
  TFloat Grayscale,
  GrayTrans;

  TTransform(TBlock * R, TBlock * D, TFloat S, TFloat G)
  { Range = R; Domain = D; Grayscale = S; GrayTrans = G; }
  virtual ~TTransform() {}

  virtual void Display();
};

enum TSymmetry {identity, reflect_y, reflect_x, rot180,
               reflect_y_eq_x, rot90, rot270, reflect_y_eq_minus_x};

class TRectTransform : public TTransform
{
public:
  TSymmetry Symmetry;

  TRectTransform(TBlock * R, TBlock * D, TFloat S, TFloat D, TSymmetry Sym) : TTransform(R,D,S,D)
  { Symmetry = Sym; }

  virtual void Display()
  { TTransform::Display();
    printf("Symmetry = %d\n",Symmetry); }
};

#if !defined (BLOCK)

BLOCK.CC

//
// BLOCK.CC
//

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "global.h"
#include "block.h"

//
// TMemScreen methods
//

int TMemScreen::AvgGray(int ulx, int uly, int wid, int ht)
{
  register int row, col, GraySum = 0;

  for (row = uly; row < uly+ht; row++)
for (col = ulx; col < ulx+wid; col++)
    GraySum += get(col,row);
return(GraySum/(wid*ht));
}

// TRectangularBlock methods
//
void TRectangularBlock::Display()
{
    printf("This block spans from (%d,%d) to (%d,%d)\n", ulx,uly,ulx+width-1,uly+height-1);
}

// TRectRangeBlock methods
//
TRectRangeBlock::TRectRangeBlock(
    int x,int y,int wid,int ht,TMemScreen * Image) :
    TRectangularBlock(x,y,wid,ht)
{
    register int  row,col,gray;

    RangeSum = RngSqrSum = 0;
    for (row = uly; row < uly+height; row++)
        for (col = ulx; col < ulx+width; col++)
            {
                gray = Image->get (col,row);
                RangeSum += gray;
                RngSqrSum += gray*gray;
            }
}

void TRectRangeBlock::Display()
{
    TRectangularBlock::Display();
    printf("Range Sum = %f  Range`2 Sum = %f\n",RangeSum,RngSqrSum);
}

TFloat TRectRangeBlock::MinimizeRMS(
    TMemScreen * Image, TBlock * Domain, TTransform * Trans)
{
    TFloat DomSum, DomSqrSum, RngDomSum,
        SDenom, S, 0, MinRMS, RMS,
        n;
    int    row, col,
        temp1, temp2,
        XDir, YDir,
        DomRow, DomCol, DomStartCol;
    TSymmetry Sym;
    TMemScreen *DomGrays;

    Trans->Range = this;
    Trans->Domain = Domain;
    MinRMS = NearlyInfinite;
    n = width*height;

    for (Sym = identity; Sym <= reflect_y_eq_minus_x; Sym++)
RngDomSum = 0;
if (Sym < reflect_y_eq_x) {
    DomGrays = ((TRectDomBlock *)Domain)->DomXisX;
    DomSum = ((TRectDomBlock *)Domain)->DomSum1;
    DomSqrSum = ((TRectDomBlock *)Domain)->DomSqrSum1;
    SDenom = ((TRectDomBlock *)Domain)->SDenom1;
} else {
    DomGrays = ((TRectDomBlock *)Domain)->DomXisY;
    DomSum = ((TRectDomBlock *)Domain)->DomSum2;
    DomSqrSum = ((TRectDomBlock *)Domain)->DomSqrSum2;
    SDenom = ((TRectDomBlock *)Domain)->SDenom2;
}
switch (Sym) {
    case identity:
    case reflect_y_eq_x:
    case reflect_x:
    case rot90:
        XDir = 1;
        DomStartCol = 0;
        break;
    default:
        XDir = -1;
        DomStartCol = width-1;
}
switch (Sym) {
    case identity:
    case reflect_y_eq_x:
    case reflect_y:
    case rot270:
        YDir = 1;
        DomRow = 0;
        break;
    default:
        YDir = -1;
        DomRow = height-1;
}
for (row = uly; row < uly+height; row++, DomRow += YDir) {
    for (col = ulx, DomCol = DomStartCol;
        col < ulx+width; col++, DomCol += XDir) {
        temp1 = DomGrays->get(DomCol,DomRow);
        temp2 = Image->get(col,row);
        RngDomSum += temp1*temp2;
    }
} 
if (SDenom != 0) {
    S = (n*RngDomSum - RangeSum*DomSum)/SDenom;
    D = (RangeSum-S*DomSum)/n;
} else {

S = 0;
0 = RangeSum/n;
}
if (fabs(0) >= 255)
    RMS = NearlyInfinite;  // skip this transform
else
    RMS = fabs((RngSqrSum+S*(S*DomSqrSum-2*RngDomSum+2*0*DomSum)+
        0*(0*n-2*RangeSum))/n);

if (RMS<MinRMS)
    {
        MinRMS = RMS;
        Trans->GrayScale = S;
        Trans->GrayTrans = 0;
        ((TRectTransform *)Trans)->Symmetry = Sym;
    }
}
return (MinRMS);

// TRectDomBlock methods
// TRectDomBlock::TRectDomBlock(
    int x, int y, int wid, int ht,
    TBlock * Range, TMemScreen * Image)
    : TRectangularBlock(x,y,wid,ht)
{
    TFloat X2X, Y2Y, X2Y, Y2X;
    int DomGray,
        row, col,
        n,
        DomRow1, DomCol1,
        DomRow2, DomCol2;
    RngWid = ((TRectangularBlock *)Range)->Get_width();
    RngHt = ((TRectangularBlock *)Range)->Get_height();
    n = RngWid*RngHt;
    X2X = (TFloat)width/(TFloat)RngWid;
    Y2Y = (TFloat)height/(TFloat)RngHt;
    X2Y = (TFloat)height/(TFloat)RngWid;
    Y2X = (TFloat)width/(TFloat)RngHt;
    DomSum1 = DomSum2 = DomSqrSum1 = DomSqrSum2 = 0;
    DomXisX = new TMemScreen(RngWid,RngHt);
    DomXisY = new TMemScreen(RngWid,RngHt);
    for (row = 0; row < RngHt; row++)
        for (col = 0; col < RngWid; col++)
            {
            DomGray = Image->AvgGray(ulx+(int)(col*X2X),uly+(int)(col*Y2Y),
                (int)X2X,(int)Y2Y);
            DomXisX->set(col,row,DomGray);
            DomSum1 += DomGray;
            DomSqrSum1 += DomGray*DomGray;
            DomGray = Image->AvgGray(ulx+(int)(row*Y2X),uly+(int)(col*X2Y),
                (int)Y2X,(int)X2Y),
        }
(int)Y2X,(int)X2Y);
DomXisY->set(col,row,DomGray);
DomSum2 += DomGray;
DomSqrSum2 += DomGray*DomGray;
}

SDenom1 = n*DomSqrSum1 - DomSum1*DomSum1;
SDenom2 = n*DomSqrSum2 - DomSum2*DomSum2;
}

void TRectDomBlock::Display()
{
TRectangularBlock::Display();
printf("DomSum1 = %.f DomSqrSum1 = %.f
DomSum2 = %.f DomSqrSum2 = %.f
DomSum1,DomSqrSum1,DomSum2,DomSqrSum2,SDenom1,SDenom2);n"
}


// TTransform methods
//
void TTransform::Display()
{
printf("Range Block :

Range->Display();
printf("Domain Block :

Domain->Display();
printf("S = %.f 0 = %.f",Grayscale,GrayTrans);
}

PARTITION.H

#ifndef PARTITION
#define PARTITION 1

// partition.h

#include "sl1ist.h"
#include "block.h"

// Partition-type constants

#define QuadTreePartition 1

// Maximum size of a quadtree block
#define MAX_QUAD_PIXELS 256

class TPartitioner
{
protected:
TMemScreen * ImagePtr;
TFloat tolerance;
TTransform * CurTrans,
virtual TBlock * NextRangeBlock() { return NULL; }
virtual void SetUpDomainBlocks(TBlock * Range) {};
virtual TBlock * NextDomainBlock() { return NULL; }
virtual int Encode(TTransform * trans, void ** buffer) { return 0; }
virtual int IsDivisible(TBlock * Range) { return 0; }
virtual void Split(TBlock * Range) {};

public:
TPartitioner(TMemScreen * image, TFloat t)
{ ImagePtr = image;
  tolerance = t;
}
virtual ~TPartitioner() {};

virtual int Get_width() { return ImagePtr->Get_width(); }
virtual int Get_height() { return ImagePtr->Get_height(); }
virtual int Get_type() { return 0; }
virtual int NextTransform(void ** BufPtr);

struct TQuadPartRec
{
  unsigned DomX : 9;
  unsigned DomY : 9;
  unsigned RngWid : 8;
  unsigned OSignBit : 1;
  unsigned sym : 3;
  unsigned fill : 2;
  short S;
  unsigned char 0;
};

class TQuadPart : public TPartitioner
{
protected:
TRectDomBlock * CurDom;
TSLL RangeList,
   DomLists;
int MaxDomWid,
   MaxDomHt;

virtual TBlock * NextRangeBlock();
virtual void SetUpDomainBlocks(TBlock * Range);
virtual TBlock * NextDomainBlock();
virtual int Encode(TTransform * trans, void ** buffer);
virtual int IsDivisible(TBlock * Range);
virtual void Split(TBlock * Range);

virtual void SetUpDomainLists();
virtual void BuildDomainLists(TBlock * Domain);
virtual void SetUpRangeBlocks(TBlock * Domain);

public:
TQuadPart(TMemScreen * image, TFloat t) : TPartitioner(image, t)
{ 
  TRectangularBlock * TempDom;
}
MinTrans = new TRectTransform(NULL, NULL, 0, 0, identity);
CurTrans = new TRectTransform(NULL, NULL, 0, 0, identity);
SetUpDomainLists();
TempDom = new TRectangularBlock(0,0,image->Get_width(),
                        image->Get_height());
SetUpRangeBlocks(TempDom);
}
virtual TQuadPart()
{
    delete MinTrans;
    delete CurTrans;
}
virtual int Get_type() { return QuadTreePartition; }
virtual void Display() { RangeList.Display(); DomLists.Display(); }
};
#endif /* PARTITION */

PARTITION.CC

#include <stdio.h>
#include <math.h>
#include <new.h>
#include "global.h"
#include "partition.h"

int TPartitioner::NextTransform(void ** BufPtr)
{
    TBlock * Range,
    * Domain;
    TTransform * TempTrans;
    TFloat RMS,
    MinRMS = NearlyInfinite;
    int FoundDom = 0,
    len;
    Range = NextRangeBlock();
    if (Range == NULL) return(0); // Out of range blocks
    SetUpDomainBlocks(Range);
    while (Domain = NextDomainBlockQ)
    {
        RMS = Range->MinimizeRMS(ImagePtr, Domain, CurTrans);
        if (RMS < MinRMS)
        {
            MinRMS = RMS;
            FoundDom = 1;
            len = Domain->Get_len();
TempTrans = MinTrans; // Swap pointers
MinTrans = CurTrans;
CurTrans = MinTrans;

FoundDom = (RMS<=tolerance);
if (FoundDom) break; // Found a good enough match
}

if (FoundDom || !IsDivisible(Range)) {
    #ifdef VERBOSE
    MinTrans->Display(); // Debugging output
    printf("RMS error = %f\n", RMS);
    #endif
    len = Encode(MinTrans, BufPtr); // Return the best match found
} else {
    Split(Range); // Divide into sub-range blocks
    len = NextTransform(BufPtr);
}

return len;

// TQuadPart methods

TBlock * TQuadPart::NextRangeBlock()
{
    TBlock * next;
    next = (TBlock *)RangeList.Head();
    if (next) RangeList.Remove(next); // Remove if not empty
    return next;
}

void TQuadPart::SetUpDomainBlocks(TBlock * Range)
{
    TSLLLItem * CurList; // Points to the domain list for this range block
    CurList = (TSLLLItem *)DomLists.Nth(((TQuadRectRangeBlock *)Range)->Get_level());
    CurDom = (TRectDomBlock *)CurList->SubList.Head();
}

TBlock * TQuadPart::NextDomainBlock()
{
    if (CurDom) CurDom = (TRectDomBlock *)CurDom->Next;
    return CurDom;
}

int TQuadPart::Encode(TTransform * trans, void ** buffer)
{
    TQuadPartRec = OutBuf;
}
OutBuf = new TQuadPartRec;

// Fill in the output record.

OutBuf->DomX = ((TRectangularBlock *) (trans->Domain))->Get_ulx();
OutBuf->DomY = ((TRectangularBlock *) (trans->Domain))->Get_uly();
OutBuf->RngWid = ((TRectangularBlock *) (trans->Range))->Get_width();
OutBuf->S = (int)(trans->GrayScale*GSMult);
OutBuf->O = (int)(fabs(trans->GrayTrans));
OutBuf->OSignBit = (trans->GrayTrans > 0)? 0 : 1; // Save a sign bit
OutBuf->Sym = (int)(((TRectTransform *) (trans))->Symmetry);

*buffer = OutBuf;
delete trans->Range; // Finished with this range block
return sizeof(TQuadPartRec);

int TQuadPart::IsDivisible(TBlock * Range)
{
    // Make sure the width and height are both even.
    return !((((TRectangularBlock *) Range)->Get_width() & 1) ||
               (((TRectangularBlock *) Range)->Get_height() & 1));
}

void TQuadPart::Split(TBlock * Range)
{
    int CurrLevel,
        RngWid, RngHt;
    register TQuadRectRangeBlock * RangePtr = (TQuadRectRangeBlock *) Range;

    RngWid = RangePtr->Get_width()>>1;
    RngHt = RangePtr->Get_height()>>1;

    CurrLevel = RangePtr->Get_level()+1;

    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx()+RngWid, RangePtr->Get_uly()+RngHt,
        RngWid, RngHt, ImagePtr, CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx(), RangePtr->Get_uly()+RngHt,
        RngWid, RngHt, ImagePtr, CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx()+RngWid, RangePtr->Get_uly(),
        RngWid, RngHt, ImagePtr, CurrLevel));
    RangeList.Insert(new TQuadRectRangeBlock(
        RangePtr->Get_ulx(), RangePtr->Get_uly(),
        RngWid, RngHt, ImagePtr, CurrLevel));
    delete Range; // Finished with this range block
}

void TQuadPart::SetUpDomainLists()
{
    TRectangularBlock * Domain;

    MaxDomWid = ImagePtr->Get_width();
    MaxDomHt = ImagePtr->Get_height();

    while (MaxDomWid*MaxDomHt > MAX_QUAD_PIXELS)
{ MaxDomWid = MaxDomWid<<1;
 MaxDomHt = MaxDomHt>>1;
}

Domain = new TRectangularBlock(0,0,MaxDomWid,MaxDomHt);
BuildDomainLists(Domain);
}

void TQuadPart::BuildDomainLists(TBlock * Domain)
{
TRectangularBlock * Range;

if (IsDivisible(Domain))
{
int row, col,
wid, ht,
XDoms, YDoms;
TSLLLItem * CurrList;

wid = ((TRectangularBlock *)Domain)->Get_width();
ht = ((TRectangularBlock *)Domain)->Get_height();
Range = new TRectangularBlock(0,0,wid>>1,ht>>1);
XDoms = ImagePtr->Get_width()/wid;
YDoms = ImagePtr->Get_height()/ht;

CurrList = new TSLLLItem();
DomLists.Append(CurrList);

for (row = 0; row < YDoms; row++)
for (col = 0; col < XDoms; col++)
CurrList->SubList.Append(
new TRectDomBlock(
    col*wid,row*ht,wid,ht,Range,ImagePtr) );

BuildDomainLists(Range);  // Build the next lower level
}

delete Domain;
}

void TQuadPart::SetUpRangeBlocks(TBlock * Domain)
{
register TRectangularBlock * DomPtr = (TRectangularBlock *)Domain;
int NewWid,
NewHt;
NewWid = DomPtr->Get_width()>>1;
NewHt = DomPtr->Get_height()>>1;

if (DomPtr->Get_width() == MaxDomWid)
{
RangeList.Insert(new TQuadRectRangeBlock(
    DomPtr->Get_ulx()+NewWid,DomPtr->Get_uly()+NewHt,
    NewWid,NewHt,ImagePtr,1));
RangeList.Insert(new TQuadRectRangeBlock(
    DomPtr->Get_ulx(),DomPtr->Get_uly()+NewHt,
    NewWid,NewHt,ImagePtr,1));
}
NewWid, NewHt, ImagePtr, 1));
RangeList.Insert(new TQuadRectRangeBlock(
    DomPtr->Get_ulx() + NewWid, DomPtr->Get_uly(),
    NewWid, NewHt, ImagePtr, 1));
RangeList.Insert(new TQuadRectRangeBlock(
    DomPtr->Get_ulx(), DomPtr->Get_uly(),
    NewWid, NewHt, ImagePtr, 1));
}
else
{
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx() + NewWid, DomPtr->Get_uly() + NewHt,
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx(), DomPtr->Get_uly() + NewHt,
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx() + NewWid, DomPtr->Get_uly(),
        NewWid, NewHt));
    SetUpRangeBlocks(new TRectangularBlock(
        DomPtr->Get_ulx(), DomPtr->Get_uly(),
        NewWid, NewHt));
}
delete Domain;

INFILE.H

#ifndef INFILE
#define INFILE 1

// infile.h

#include <stdio.h>
#include "block.h"

class TInputFile
{
    protected:
        TMemScreen * ImagePtr;

    public:
        TInputFile() { ImagePtr = NULL; }
        virtual ~TInputFile() { delete ImagePtr; }

        virtual TMemScreen * Get_ImagePtr() { return ImagePtr; }
        virtual int Get_width() { return 0; }
        virtual int Get_height() { return 0; }
};

struct TTargaHeader
{
    // Assumes sizeof(short)==2
    char IDFieldLength;
    char ColorMapType;
    char ImageType;
short FirstColorMapEntry;
short ColorMapLength;
char ColorMapEntrySize;
short ImageXOrigin;
short ImageYOrigin;
short ImageWidth;
short ImageHeight;
char BitsPerPixel;
char ImageDescriptorBits;
};

class TTgaFile : public TInputFile
{
protected:
    TTargaHeader header;
    FILE * fp;
    char * buffer;

    virtual void ReadTargaHeader();
    virtual void GetRow();

public:
    TTgaFile(char * name);
    virtual ~TTgaFile()
    {
        delete buffer;
        delete ImagePtr;
    }

    virtual Get_width() { return header.ImageWidth; }
    virtual Get_height() { return header.ImageWidth; }
};

#endif /*  INFILE */

INFILE.CC

//
// INFILE.CC
//

#include <stdio.h>
#include <new.h>
#include "infile.h"

//
// TTgaFile methods
//

void TTgaFile::ReadTargaHeader()
{
    if (fread(&header, sizeof(header), 1, fp) == 0)
    {
        perror("Error reading the Targa header");
        exit(1);
    }
void TTgaFile::GetRow()
{
    if (fread(buffer, header.ImageWidth, 1, fp) == 0)
    {
        perror("Error reading a Targa row");
        exit(1);
    }
}

TTgaFile::TTgaFile(char * name)
{
    int DestRow,
    SrcRow,
    SrcCol,
    next;

    if ((fp = fopen(name,"r")) == NULL)
    {
        perror("Error opening the Targa file");
        exit(1);
    }
    ReadTargaHeader();
    ImagePtr = new TMemScreen(header.ImageWidth, header.ImageHeight);
    buffer = new char[header.ImageWidth];

    switch (header.ImageDescriptorBits & 0x30)
    {
    case 32 : DestRow = 0;
                next = 1;
                break;
    case 0  : DestRow = header.ImageHeight-1;
                next = -1;
                break;
    default : perror("Right-to-left encoding not supported");
              exit(1);
    }

    for (SrcRow = 0; SrcRow<header.ImageHeight; SrcRow++)
    {
        GetRow();
        for (SrcCol = 0; SrcCol<header.ImageWidth; SrcCol++)
            ImagePtr->set(SrcCol, DestRow, buffer[SrCcol]);

        DestRow += next;
    }

    if (fclose(fp) == EOF)
    {
        perror("Error closing the Targa file");
        exit(1);
    }
}
#ifndef ENCODE
#define ENCODE 1
//
// encode.h
//

#include <stdio.h>
#include "block.h"
#include "partition.h"

#define IdentityEncode 1
#define LZWEncode 2

struct TEncodeHeader {
    unsigned short width,
    height;
    unsigned char PartType,
    EncType;
};

class TEncoder {
protected:
    int EncodeType;
    FILE * fp;

    virtual void WriteHeader(TPartitioner * Part);
    virtual void WriteToFile(int len, void * buffer);
    virtual void CloseOutputFile();

public:
    TEncoder(char * name) {
        if ((fp = fopen(name,"w")) == NULL) {
            perror("Error opening the output file");
            exit(1);
        }
        EncodeType = IdentityEncode;
    }
    virtual ^TEncoder() { CloseOutputFile(); }

    virtual void EncodeFIF(TPartitioner * Part);
};

#endif /*  ENCODE */

ENCODE.CC

//
// ENCODE.CC
//

#include <stdio.h>
#include <new.h>
#include "encode.h"
#include "partition.h"

//
// TEncoder methods
//
void TEncoder::WriteHeader(TPartitioner * Part) {
  TEncodeHeader header;
  header.width = Part->Get_width();
  header.height = Part->Get_height();
  header.PartType = Part->Get_type();
  header.EncType = EncodeType;
  if (fwrite((char *)&header, sizeof(header), 1, fp) == 0) {
    perror("Error writing to the header");
    exit(1);
  }
}

void TEncoder::WriteToFile(int len, void * buffer) {
  if (fwrite((char *)buffer, len, 1, fp) == 0) {
    perror("Error writing to the output file");
    exit(1);
  }
}

void TEncoder::CloseOutputFile() {
  if (fclose(fp) == EOF) {
    perror("Error closing to the output file");
    exit(1);
  }
}

void TEncoder::EncodeFIF(TPartitioner * Part) {
  int len;
  void * buffer;
  WriteHeader(Part);
  while (len = Part->NextTransform(&buffer)) {
    WriteToFile(len, buffer);
    delete buffer;
  }
}

FRACDCMP.H

#ifndef FRACDCMP
#define FRACDCMP 1
/ // fracdcmp.h 
//
#include <stdio.h>
#include "block.h"
#include "sllist.h"
#include "infile.h"
#include "encode.h"

class TTransNode : public TSLLItem 
{ 
protected: 
TBlock * Range, 
* Domain; 
TFloat s, o; 

public: 
TTransNode(TBlock * Rng, TBlock * Dom, TFloat GScale, TFloat GTrans) : 
TSLLItem() 
{ Range = Rng; Domain = Dom; s = GScale; o = GTrans; } 
virtual ~TTransNode() 
{ delete Range; 
delete Domain; } 

virtual void Display() 
{ 
printf("Range Block :\n"); 
Range->Display(); 
printf("Domain Block :\n"); 
Domain->Display(); 
printf("S = %f  O = %f\n", s, o); 
} 
virtual void apply(TMemScreen * InImage, TMemScreen * OutImage) {} 
};

class TRectTransNode : public TTransNode 
{ 
protected: 
TSymmetry Symmetry; 
int XStep, YStep, 
XDir, YDir, 
DomRowStart, DomColStart; 

public: 
TRectTransNode(TBlock * Rng, TBlock * Dom, TFloat GScale, TFloat GTrans, 
TSymmetry Sym) : TTransNode(Rng, Dom, GScale, GTrans) 
{ Symmetry = Sym; 
switch(Sym) 
{ 
 case identity: 
 case reflect_y: 
 case reflect_x: 
 case rot180: 
 XStep = ((TRectangularBlock *)Domain)->Get_width()/ 
 ((TRectangularBlock *)Range)->Get_width(); 
 YStep = ((TRectangularBlock *)Domain)->Get_height()/ 
 ((TRectangularBlock *)Range)->Get_height(); 
 break;
default:  XStep = ((TRectangularBlock *)Domain)->Get_width() / ((TRectangularBlock *)Range)->Get_height();
    YStep = ((TRectangularBlock *)Domain)->Get_height() / ((TRectangularBlock *)Range)->Get_width();
}

switch(Sym) {
    case identity:
    case reflect_y:
    case rot270:
    case reflect_y_eq_x:
        XDir = 1;
        DomColStart = ((TRectangularBlock *)Domain)->Get_ulx();
        break;
    default:  XDir = -1;
        DomColStart = ((TRectangularBlock *)Domain)->Get_ulx() + ((TRectangularBlock *)Domain)->Get_width() - XStep;
    }

switch(Sym) {
    case identity:
    case reflect_x:
    case rot90:
    case reflect_y_eq_x:
        YDir = 1;
        DomRowStart = ((TRectangularBlock *)Domain)->Get_uly();
        break;
    default:  YDir = -1;
        DomRowStart = ((TRectangularBlock *)Domain)->Get_uly() + ((TRectangularBlock *)Domain)->Get_height() - YStep;
    }
}

virtual void Display()
{
    TTransNode::Display();
    printf("Symmetry = %d\n",Symmetry);
    printf("XStep=%d, XDir=%d, YStep=%d, YDir=%d\n",XStep,XDir,YStep,YDir);
    printf("DomCol = %d, DomRow = %d\n",DomColStart,DomRowStart);
}

virtual void apply(TMemScreen * InImage, TMemScreen * OutImage);
};

class TTransList : public TSLL
{
    protected:
    FILE * fp;
    TEncodeHeader header;

    public:
    TTransList(char * name)
    {
        if ((fp = fopen(name,"r")) == NULL)
        {
            perror("Error opening the input file");
            exit(1);
        }
    }
else {
    if (fread(&header, sizeof(header), 1, fp) == 0) {
        perror("Error reading the FIF header");
        exit(1);
    }
}
}
virtual TTTransListO { fclose(fp); }

virtual void ReadTransformsO {}
virtual void apply(TMemScreen * InImage, TMemScreen * OutImage) {
    TTransNode * CurTrans;
    CurTrans = (TTransNode *)HeadPtr;
    while (CurTrans != NULL) {
        CurTrans->apply(InImage, OutImage);
        CurTrans = (TTransNode *)CurTrans->Next;
    }
}
virtual int Get_widthO { return header.width; }
virtual int Get_heightO { return header.height; }
};

class TRectTransList : public TTTransList {
    protected:
        TQuadPartRec CurTrans;

    virtual void ReadATransformO {}{
        if (fread(&CurTrans, sizeof(CurTrans), 1, fp) == 0) {
            perror("Error reading the input file");
            exit(1);
        }
    }
    virtual void QuadReadTrans(int ulx, int uly, int wid, int ht);

    public:
        TRectTransList(char * name) : TTTransList(name) {} 
    virtual void ReadTransformsO {}
        ReadATransformO();
        QuadReadTrans(0,0,header.width,header.height);
    }
};

class TDoutTga {
    protected:
        FILE * fp;

    virtual void WriteHeader(TMemScreen * Image)
TTarget header;

header.IDFieldLength = 0;
header.ColorMapType = 0;
header.ImageType = 3;
// Color map
header.FirstColorMapEntry = 0;
header.ColorMapLength = 0;
header.ColorMapEntrySize = 0;
// Image data
header.ImageXOrigin = 0;
header.ImageYOrigin = 0;
header.ImageWidth = Image->Get_width();
header.ImageHeight = Image->Get_height();
header.BitsPerPixel = 8;
header.ImageDescriptorBits = 0x20;

fwrite(&header, sizeof(header), 1, fp);
}

virtual void WriteBody(TMemScreen * Image)
{
    int wid, ht, row, col;
    unsigned char * rowbuf;

    wid = Image->Get_width();
    ht = Image->Get_height();
    rowbuf = new unsigned char[wid];

    for (row=0; row<ht; row++)
    {
        for (col=0; col<wid; col++)
        {
            rowbuf[col] = Image->get(col, row);
            fwrite(rowbuf, wid, 1, fp);
        }
    }
    delete rowbuf;
}

public:
T0utTga(char * name, TMemScreen * Image)
{
    if ((fp = fopen(name, "w")) == NULL)
    {
        perror("Error opening the output file");
        exit(1);
    }
    else
    {
        WriteHeader(Image);
        WriteBody(Image);
        if (fclose(fp) == EOF)
        {
            perror("Error opening the output file");
            exit(1);
        }
    }
}
virtual ~TOutTga() {};
#endif /* FRACDCMP */

FRACDCMP.CC

//
// FRACDCMP.CC
//

#include <stdio.h>
#include "block.h"
#include "encode.h"
#include "fracdcmp.h"

void TRectTransNode::apply(TMemScreen * Inlmage, TMemScreen * Outlmage) {

  int temp,
  RngXStart,
  RngWid,
  RngYStart,
  RngHt,
  row, col,
  DomRow,
  DomCol,
  RngPos;

  RngXStart = ((TRectangularBlock *)Range)->Get_ulx();
  RngWid = ((TRectangularBlock *)Range)->Get_width();
  RngYStart = ((TRectangularBlock *)Range)->Get_uly();
  RngHt = ((TRectangularBlock *)Range)->Get_height();

  switch(Symmetry) {
  case identity:
  case reflect_y:
  case reflect_x:
  case rot180:
    for (row = 0; row < RngHt; row++)
      for (col = 0; col < RngWid; col++) {
        temp = (int)(s*Inlmage->AvgGray(DomColStart+(int)(col*XStep*XDir),
        DomRowStart+(int)(row*YStep*YDir),
        (int)XStep,(int)YStep)+o);

        if (temp>255)
          temp = 255;
        else if (temp < 0)
          temp = 0;
        Outlmage->set(RngXStart+col,RngYStart+row,temp);
      }
    break;
  default:
    for (row = 0; row < RngHt; row++)
      for (col = 0; col < RngWid; col++) {

temp = (int)(s*InImage->AvgGray(DomColStart+(int)(row*XStep*XDir),
  DomRowStart+(int)(col*YStep*YDir),
  (int)XStep,(int)YStep)+o);
if (temp>255)
  temp = 255;
else if (temp < 0)
  temp = 0;
OutImage->set(RngXStart+col,RngYStart+row,temp);
}
}

void TRectTransList::QuadReadTrans(int ulx, int uly, int wid, int ht)
{
  if (CurTrans.RngWid == wid)
  {
    TFloat TempO;
    TempO = CurTrans.O;
    if (CurTrans.OSignBit & 1)
      TempO = -TempO; // Negative bit was set
    Insert(new TRectTransNode(  
      new TRectangularBlock(ulx,uly,wid,ht),
      new TRectangularBlock(CurTrans.DomX, CurTrans.DomY,
        wid«1, ht«1),
      (float)CurTrans.S/GSMult,TempO,(TSymmetry)(CurTrans.sym)));
    #ifdef VERBOSE
    HeadPtr->Display();
    #endif
  } else {
    int NewWid, NewHt;
    NewWid = wid»1;
    NewHt = ht»1;
    QuadReadTrans(ulx,uly,NewWid,NewHt);
    ReadATransform();
    QuadReadTrans(ulx+NewWid,uly,NewWid,NewHt);
    ReadATransform();
    QuadReadTrans(ulx,uly+NewHt,NewWid,NewHt);
    ReadATransform();
    QuadReadTrans(ulx-NewWid,uly+NewHt,NewWid,NewHt);
  }
}

int main(int argc, char *argv[])
{
  int wid, ht, row, col, iter, num_iter = 10;
  TRectTransList * TransList;
  TMemScreen * * InImage,
  * OutImage,
if (argc < 3 || argc > 4)
   fprintf(stderr,"Usage : %s fif-file tga-file [iterations]\n",argv[0]);
else {
   strcpy(fif_file,argv[1]);
   strcat(fif_file,".FIF");

   strcpy(tga_file,argv[2]);
   strcat(tga_file,".TGA");

   if (argc == 4)
      sscanf(argv[3],"%d",&num_iter);

   TransList = new TRectTransList(fif_file);
   TransList->ReadTransforms();
   wid = TransList->Get_width();
   ht = TransList->Get_height();
   Inlmage = new TMemScreen(wid, ht);
   Outlmage = new TMemScreen(wid, ht);
   for (row=0; row<ht; row++)
      for (col=0; col<wid; col++)
         Inlmage->set(col,row,128);
   for (iter=0; iter<num_iter; iter++)
   {
      TransList->apply(Inlmage, Outlmage);
      TmpImage = Inlmage;
      Inlmage = OutImage;
      OutImage = TmpImage;
   }
   TgaFile = new TOutTga(tga_file, Inlmage);
   delete TgaFile;
   delete Inlmage;
   delete OutImage;
   delete TransList;
}

return(1);
}

Turbo Pascal Source

FRACTIN.PAS

unit FractIn;

interface

uses
   Crt,
   Dos,
   CSVGA256,
const
ScreenWidth = 320;
ScreenWidth = 200;
GSMult = 512; { Amount by which to multiply the "S" values }
{ when converting to a integer }

type
TFloat =
{$IFDEF N+}
double
{$ELSE}
real
{$ENDIF}

PRow = TRow;
TRow = { The grayscale values for a row }
array[0..ScreenWidth-1] of byte; { of pixels. }
PScreen = TScreen;
TScreen = { The grayscale values for a full }
array[0..ScreenHeight-1] of TRow; { screen of pixels. }
PScreenView = TScreenView;
TScreenView = { Used to represent a rectangular }
array[0..$FFFF-1] of byte; { chunk of the screen }
PTransform = TTransform;
PImage = TImage;
PBlock = TBlock;
TBlock = { Generic range/domain block }
object
constructor Init;
destructor Done; virtual;
procedure Display; virtual;
function MinimizeRMS(Image : PImage; Domain : PBlock;
var Trans : PTransform) : TFloat; virtual;
end; { TBlock }

TRectangularBlock = TRectangularBlock;
TRectangularBlock = { Rectangular block }
object(TBlock)
ulx,uly,
width,
height : word;

constructor Init(x,y,wid,ht : word);
procedure Display; virtual;
end; { TRectangularBlock }

PRectRangeBlock = TRectRangeBlock;
TRectRangeBlock = { Rectangular range block }
object(TRectangularBlock)
RangeSum, { Used in RMS calculations }
RngSqrSum : TFloat;

constructor Init(x,y,wid,ht : word; Image : PImage);
procedure Display; virtual;
function MinimizeRMS(Image : PImage; Domain : PBlock;
var Trans : TTransform) : TFloat; virtual;
end; { TRectRangeBlock }

PRectDomBlock = TRectDomBlock;
TRectDomBlock = { Rectangular domain block }
object(TRectangularBlock)
RngWid,
RngHt : word;
DomXisX, { Used in RMS calculations }
DomXisY : PScreenView;
DomSum1,
DomSum2,
DomSqrSum1,
DomSqrSum2 : TFloat;

constructor Init(x,y,wid,ht : word; Range : PBlock; Image : PImage);
destructor Done; virtual;
end; { TRectDomBlock }

TSymmetry = 0..7;
TTransform = { Generic local IFS transform }
object
Range,
Domain : PBlock;
Grayscale,
GrayTrans : TFloat;

constructor Init(R,D : PBlock; S,O : TFloat);
destructor Done; virtual;
procedure Display; virtual;
end; { TTransform }

PTransform = TRectTransform;
TRectTransform = { Local IFS with rectangular blocks }
object(TTransform)
Symmetry : TSymmetry;

constructor Init(R,D : PBlock; S,O : TFloat; Sym : TSymmetry);
procedure Display; virtual;
end;
TImage = { Generic image-to-be-compressed }
object
Width,
Height : word;
MemScreen : PScreenView;

constructor Init;
destructor Done; virtual;
procedure Display(DoWait : boolean); virtual;
function AvgGray(ulx,uly,wid,ht : word) : word;
end; { TImage }

PTgaImage =
```
^TTgImage;
TTgImage = {  Targa input file }
  object(TImage)
    Header : TTargaHeader;
    constructor Init(TgaFile : string);
      procedure Display(DoWait : boolean); virtual;
    end; { TTgImage }

function DefaultType(Name : string; FType : string) : PathStr;
  { Returns Name+FType if Name does }
  { not already have a file type. }
  { Otherwise returns Name }
{$IFDEF logging}
var
  logfile : text;
{$ENDIF}
implementation

var
  ActualScreen : TScreen absolute $A000 : $0000;

procedure SetGrayPal;
var
  GrayPal : TVGAPalette;
  Color : byte;
begin { SetGrayPal }
  for Color := 0 to 63 do
  begin
    GrayPal[4 * Color].r := Color;
    GrayPal[4 * Color].g := Color;
    GrayPal[4 * Color].b := Color;
    GrayPal[4 * Color + 1] := GrayPal[4 * Color];
    GrayPal[4 * Color + 2] := GrayPal[4 * Color];
    GrayPal[4 * Color + 3] := GrayPal[4 * Color];
  end;
  VGASetAllPalette(GrayPal);
end; { SetGrayPal }

{<<<<<<<<<<<<<<<<<<<<<<<< TBlock methods »»»»»»»»»»»»»}

constructor TBlock.Init;
begin { TBlock.Init }
end; { TBlock.Init }

destructor TBlock.Done;
begin { TBlock.Done }
end; { TBlock.Done }

procedure TBlock.Display;
begin { TBlock.Display }
  RunError(211); {  Abstract object method }
end; { TBlock.Display }

function TBlock.MinimizeRMS(Image : PImage; Domain : PBlock;
  var Trans : PTransform) : TFloat;
```
begin { TBlock.MinimizeRMS }
  RunError(211); { Abstract object method }
end; { TBlock.MinimizeRMS }

{<<<<<<<<<<<<<<<<<<<<<<<<<<< TRectangularBlock methods >>>>>>>>>>>>>>>>>>>>>>>>>>>>

constructor TRectangularBlock.Init(x,y,wid,ht : word);
begin { TRectangularBlock.Init }
  if not TBlock.Init then fail;

  ulx := x;
  uly := y;
  width := wid;
  height := ht;
end; { TRectangularBlock.Init }

procedure TRectangularBlock.Display;
begin { TRectangularBlock.Display }
  writeln('This block spans from (',ulx,',',uly,') to (',ulx+width-l,'',',uly+height-l,')');
  {$IFDEF logging}
  writeln(logfile,'This block spans from (',ulx,',',uly,') to (',ulx+width-l,'
',',',uly+height-l,')');
  {$ENDIF}
end; { TRectangularBlock.Display }

{<<<<<<<<<<<<<<<<<<<<<<<<<<< TRectRangeBlock methods >>>>>>>>>>>>>>>>>>>>>>>>>>>>

constructor TRectRangeBlock.Init(x,y,wid,ht : word; Image : PImage);
begin { TRectangularBlock.Init >
  if not TRectangularBlock.Init(x,y,wid,ht) then fail;

  RangeSum := 0;
  RngSqrSum := 0;
  for row := uly to uly+height-1 do
    for col := ulx to ulx+width-1 do
      with Image" do
        begin
          Gray := MemScreen^[row*width+col];
          RangeSum := RangeSum+Gray;
          RngSqrSum := RngSqrSum+sqr(Gray);
        end;
  end; { TRectangularBlock.Init }

procedure TRectRangeBlock.Display;
begin { TRectangularBlock.Display }
  TRectangularBlock.Display;
  writeln('Range Sum = ',RangeSum,' Range"2 Sum = ',RngSqrSum);
  {$IFDEF logging}
  writeln(logfile,'Range Sum = ',RangeSum,' Range~2 Sum = ',RngSqrSum);
  {$ENDIF}
end; { TRectRangeBlock.Display }
function TRectRangeBlock.MinimizeRMS(
  Image : PImage; Domain : PBlock;
  var Trans : PTransform) : TFloat;
var
  n : longint;
  DomSum,
  DomSqrSum,
  RngDomSum : TFloat;
  row, col,
  Temp1,
  Temp2,
  DomPos,
  RngPos : word;
  SDenom,
  S, 0,
  MinRMS,
  RMS : TFloat;
  Sym : TSymmetry;
  DomGrays : PScreenView;
begin { TRectRangeBlock.MinimizeRMS }
  Trans.Range := Addr(Self);
  Trans.Domain := Domain;
  MinRMS := NearlyInfinite;
  n := Width*Height;

  for Sym := 0 to 7 do
    begin
      RngDomSum := 0;
      if (Sym < 4) then with PRectDomBlock(Domain) do
        begin
          DomGrays := DomXisX;
          DomSum := DomSum1;
          DomSqrSum := DomSqrSum1;
        end
      else with PRectDomBlock(Domain) do
        begin
          DomGrays := DomXisY;
          DomSum := DomSum2;
          DomSqrSum := DomSqrSum2;
        end;
      SDenom := (n*DomSqrSum-sqr(DomSum));
      with Image do
        begin
          RngPos := uly*Width+ulx;
          case Sym of
            0,4 : { Identity or reflect about the line y=x }
              for row := 0 to Self.Height-1 do
                begin
                  DomPos := row*Self.Width;
                  for col := 0 to Self.Width-1 do
                    begin
                      Temp1 := DomGrays[DomPos+col];
                      Temp2 := MemScreen[RngPos+col];
                      RngDomSum := RngDomSum + Temp1*Temp2;
                    end;
                  RngPos := RngPos+Width;
                end;
          end;
        end;
    end;
end;
1, 5: { reflect about the y-axis or }
{ rotate by 90 degrees }
for row := 0 to Self.Height-1 do
begin
    DomPos := row*Self.Width;
    for col := 0 to Self.Width-1 do
    begin
        Temp1 := DomGrays[DomPos+Self.Width-col-1];
        Temp2 := MemScreen[RngPos+col];
        RngDomSum := RngDomSum + Temp1*Temp2;
    end;
    RngPos := RngPos+Width;
end;
2, 6: { reflect about the x-axis or }
{ rotate by 270 degrees }
for row := Self.Height-1 downto 0 do
begin
    DomPos := row*Self.Width;
    for col := 0 to Self.Width-1 do
    begin
        Temp1 := DomGrays[DomPos+col];
        Temp2 := MemScreen[RngPos+col];
        RngDomSum := RngDomSum + Temp1*Temp2;
    end;
    RngPos := RngPos+Width;
end;
3, 7: { rotate by 180 degrees or }
{ reflect about the line y=-x }
for row := Self.Height-1 downto 0 do
begin
    DomPos := row*Self.Width;
    for col := 0 to Self.Width-1 do
    begin
        Temp1 := DomGrays[DomPos+Self.Width-col-1];
        Temp2 := MemScreen[RngPos+col];
        RngDomSum := RngDomSum + Temp1*Temp2;
    end;
    RngPos := RngPos+Width;
end; { End of cases }
end; { End of with }

if (SDenom<>0)
then begin
    S := (n*RngDomSum-RangeSum*DomSum)/SDenom;
    0 := (RangeSum-S*DomSum)/n;
end
else begin
    S := 0;
    0 := RangeSum/n;
end;
if (abs(0) >= 256)
then RMS := NearlyInfinite { skip this transform }
else RMS := (RngSqrSum+S*(S*DomSqrSum-2*RngDomSum+2*0*DomSum)+
0*(0*n-2*RangeSum))/n;

if (RMS<MinRMS)
then begin
    MinRMS := RMS;
    Trans^GrayScale := S;
    Trans^GrayTrans := 0;
    PRectTransform(Trans)^Symmetry := Sym;
end;
end; { End of symmetry loop }
MinimizeRMS := MinRMS;
end; { TRectRangeBlock.MinimizeRMS }

constructor TRectDomBlock.Init(x,y,wid,ht : word; Range : PBlock;
    Image : PImage);
var
    DomGray,
    X2X,Y2Y,
    X2Y,Y2X,
    row,
    col : word;
begin { TRectDomBlock.Init }
    if not TRectangularBlock.Init(x,y,wid,ht)
        then fail;
    RngWid := PRectangularBlock(Range)^Width;
    RngHt := PRectangularBlock(Range)^Height;
    X2X := Width div RngWid;
    Y2Y := Height div RngHt;
    X2Y := Height div RngWid;
    Y2X := Width div RngHt;
    DomSum1 := 0;
    DomSqrSum1 := 0;
    DomSum2 := 0;
    DomSqrSum2 := 0;
    GetMem(DomXisX,RngHt*RngWid);
    GetMem(DomXisY,RngHt*RngWid);
    for row := 0 to RngHt-1 do
        for col := 0 to RngWid-1 do
            with Image^ do
                begin
                    DomGray := AvgGray(col*X2X+ulx,row*Y2Y+uly,X2X,Y2Y);
                    DomXisX[row*RngWid+col] := DomGray;
                    DomSum1 := DomSum1+DomGray;
                    DomSqrSum1 := DomSqrSum1+sqr(DomGray);
                    DomGray := AvgGray(row*Y2X+ulx,col*X2Y+uly,Y2X,X2Y);
                    DomXisY[row*RngWid+col] := DomGray;
                    DomSum2 := DomSum2+DomGray;
                    DomSqrSum2 := DomSqrSum2+sqr(DomGray);
                end;
end; { TRectDomBlock.Init }

destructor TRectDomBlock.Done;
begin { TRectDomBlock.Done }
    FreeMem(DomXisX,RngHt*RngWid);
    FreeMem(DomXisY,RngHt*RngWid);
    TRectangularBlock.Done;
constructor TTransform.Init(R, D : PBlock; S, O : TFloat);
begin { TTransform.Init }
  Range := R;
  Domain := D;
  GrayScale := S;
  GrayTrans := 0;
end; { TTransform.Init }

destructor TTransform.Done;
begin { TTransform.Done }
end; { TTransform.Done }

procedure TTransform.Display;
begin { TTransform.Display }
  writeln('Range block :');
  {$IFDEF logging}
  writeln(logfile,'Range block :');
  {$ENDIF}
  Range^.Display;
  writeln('Domain block :');
  {$IFDEF logging}
  writeln(logfile,'Domain block :');
  {$ENDIF}
  Domain^.Display;
  writeln('S = ',GrayScale,' 0=',GrayTrans);
  {$IFDEF logging}
  writeln(logfile,'S = ',GrayScale,' 0=',GrayTrans);
  {$ENDIF}
end; { TTransform.Display }

constructor TRectTransform.Init(R, D : PBlock; S, O : TFloat; Sym : TSymmetry);
begin { TRectTransform.Init }
  if not TTransform.Init(R, D, S, O)
  then fail;
  Symmetry := Sym;
end; { TRectTransform.Init }

procedure TRectTransform.Display;
begin { TRectTransform.Display }
  TTransform.Display;
  writeln('Symmetry=',Symmetry);
  {$IFDEF logging}
  writeln(logfile,'Symmetry=',Symmetry);
  {$ENDIF}
end; { TRectTransform.Display }

constructor TImage.Init;
begin { TImage.Init }
end; { TImage.Init }
destructor TImage.Done;
var
  count : word;
begin { TImage.Done }
  FreeMem(MemScreen,Height*Width);
end; { TImage.Done }

procedure TImage.Display(DoWait : boolean);
var
  Row,
  RowOffset,
  ColOffset : word;
begin { TImage.Display }
  SetVGAMode320By200;
  SetGrayPal;
  FillChar(ActualScreen,ScreenWidth*ScreenHeight,0); { Clear the screen }
  RowOffset := (ScreenHeight - Height) div 2;
  ColOffset := (ScreenWidth - Width) div 2;
  for Row := 0 to Height-1 do
    Move(MemScreen[Row*Width], ActualScreen[Row + RowOffset, ColOffset], Width);
  if DoWait
    then begin
      Wait;
      CloseVGA256;
    end;
end; { TImage.Display }

function TImage.AvgGray(ulx,uly,wid,ht : word) : word;
var
  row, col,
  GraySum : word;
begin { TImage.AvgGray }
  GraySum := 0;
  for row := uly to uly+ht-1 do
    for col := ulx to ulx+wid-1 do
      GraySum := GraySum+MemScreen[row*Width+col];
  AvgGray := GraySum div (ht*wid);
end; { TImage.AvgGray }

{<<<<<<<<<<<<<<<<<<<<<<<< TTgalmage methods »»»»»»»»»>»»}
constructor TTgalmage.Init(TgaFile : string);
var
  InFile : file;
  FileName : PathStr;
  Next : integer;
  SrcRow,
  DestRow : word;
begin { TTgalmage.Init }
  FileName := DefaultType(TgaFile,'.TGA');
  Assign(InFile, FileName);
  Reset(InFile, 1);
ReadTargaHeader(Header, InFile);
Height := Header.ImageHeight;
Width := Header.ImageWidth;
GetMem(MemScreen, Height * Width);

case (Header.ImageDescriptorBits and $30) of
  32 : begin { Untested as of yet. }
    DestRow := 0;
    Next := 1;
    end;
  0 : begin
    DestRow := Height - 1;
    Next := -1;
    end;
  else Error('Oh yuk!!!! Right-to-left encoding.', '');
end;
for SrcRow := 0 to Height - 1 do
begin
  BlockRead(InFile, MemScreen[DestRow*Width], Width);
  DestRow := DestRow + Next;
end;
Close(InFile);
end;

procedure TTgaImage.Display(DoWait : boolean);
begin { TTgaImage.Display }
  if DoWait
  then begin
    ClrScr;
    ShowTargaHeader(Header);
    Wait;
    end;
  TImage.Display(DoWait);
end; { TTgaImage.Display }

function DefaultType(Name : string; FType : String) : PathStr;
var
  DotPos, SlashPos : word;
begin { DefaultType }
  DotPos := Pos('.', Name);
  if (DotPos = 0)
  then DefaultType := Name + FType
  else begin { Make sure this '.' is not part of a }
    SlashPos := Pos('\', Name); { path name like \dir\name.typ }
    if (DotPos > SlashPos)
      then DefaultType := Name
    else begin
      DotPos := Pos('.', Copy(Name, SlashPos, Length(Name) - SlashPos + 1));
      if (DotPos = 0)
        then DefaultType := Name + FType
      else DefaultType := Name;
    end;
  end;
end;
end; { DefaultType } 
end. { FractIn } 

FRACTOUT.PAS

unit FractOut;

interface

uses 
FractIn,
Dos,
CSUtil,
SLList;

const
SimplePartition = 1;
QuadTreePartition = 2;
HVPartition = 3;

type
PPartitioner =
  "TPartitioner;
TPartitioner =
  object
    Image : PImage;
  constructor Init;
    { Descendant methods should }
    { allocate and initialize Image } 
  destructor Done; virtual; 
    { Deallocate the image } 
  procedure Compress(FileName : string; Tolerance : real); virtual;
    { Partition the image into range }
    { blocks and write the compressed } 
    { version. } 
  private
    procedure WriteHeader(var f : file); virtual;
      { Write out a header that tells the } 
      { image size }
    procedure AllocateTransforms(var CurrTrans : PTransform;
      var MinTrans : PTransform); virtual;
      { Allocate two of the right type }
      { of transaction objects for }
      { compress to use }
    procedure DeallocateTransforms(CurrTrans : PTransform;
      MinTrans : PTransform); virtual;
      { Deallocate the two working }
      { transforms }
    procedure SetUpRangeBlocks; virtual;
      { Set up the initial candidates for }
      { range blocks. }
    function NextRangeBlock : PBlock; virtual;
      { Get the next available range }
      { block (returns nil when out of }
      { range blocks) }
procedure SetUpDomainBlocks(Range : PBlock); virtual;
    { Set up the possible domain blocks }
    { for a range block }

function NextDomainBlock : PBlock; virtual;
    { Get the next domain block for the }
    { current range block (Returns nil }
    { when out of domain blocks) }

procedure WriteTransform(var f : file; Trans : PTransform); virtual;
    { Write out the necessary info to }
    { describe Trans. }

function IsDivisible(Range : PBlock) : boolean; virtual;
    { Can this range block be divided }
    { into smaller range blocks? }

procedure Split(Range : PBlock); virtual;
    { Partition a range block }

end; ( TPartitioner )

PBlockArray =
    'TBlockArray;
TBlockArray =
    array [1..$FFFF div SizeOf(PBlock)] of PBlock;
TSimplePartRec =
    record
        DomX, DomY,
        Sym : byte;
        S, O : shortint;
    end;
PSimplePart =
    'TSimplePart;
TSimplePart =
    object(TPartitioner)
        SPRWidth,
        SPRHeight,
        SPDWidth,
        SPDHeight,
        RangeNum,
        DomNum,
        MaxRanges,
        MaxDoms : word;
        RangeList,
        DomList : PBlockArray;
    constructor Init(RngWid, RngHt, DomWid, DomHt : word);
        { Descendant methods should }
        { allocate and initialize Image }
procedure Compress(FileName : string; Tolerance : real); virtual;
        { Set up the range and domain }
        { lists and call the base }
        { compress }
private
    procedure WriteHeader(var f : file); virtual;
        { Write out a header that tells the }
        { image size and partitioning }
        { scheme used. }
    procedure AllocateTransforms(var CurrTrans : PTransform;
        var MinTrans : PTransform); virtual;
        { Allocate two of the right type }
        { of transaction objects for }
        { compress to use }
procedure DeallocateTransforms(CurrTrans : PTransform; 
    MinTrans : PTransform); virtual;
    { Deallocate the two working 
    { transforms } }

procedure SetUpRangeBlocks; virtual;
    { Set up the initial candidates for } 
    { range blocks. }

function NextRangeBlock : PBlock; virtual;
    { Get the next available range } 
    { block (Returns nil when out of } 
    { range blocks) }

procedure SetUpDomainBlocks(Range : PBlock); virtual;
    { Set up the possible domain blocks } 
    { for a range block }

function NextDomainBlock : PBlock; virtual;
    { Get the next domain block for the } 
    { current range block (Returns nil } 
    { when out of domain blocks) }

procedure WriteTransform(var f : file; Trans : PTransform); virtual;
    { Write out the necessary info to } 
    { describe Trans. }

function IsDivisible(Range : PBlock) : boolean; virtual;
    { Always returns false }

end; { TSimplePart }

PSimpleTgaPart = 
    "TSimpleTgaPart; 
TSimpleTgaPart = 
    object(TSimplePart)
        constructor Init(TgaFile : string;
            RngWid, RngHt, DomWid, DomHt : word);
    end; { TSimpleTgaPart }

PBlockNode = 
    "TBlockNode; 
TBlockNode = 
    object(TSLLItem)
        BlockPtr : PBlock;

        constructor Init( Block : PBlock );
        destructor Done; virtual;
        procedure Display; virtual;
    end; { TBlockNode }

PRangeBlockNode = 
    "TRangeBlockNode; 
TRangeBlockNode = 
    object(TBlockNode)
        Level : word;

        constructor Init( Block : PBlock; TreeLevel : word );
    end; { TRangeBlockNode }

PSLLLItem = 
    "TSLLLItem; 
TSLLLItem = 
    object(TSLLLItem)
        SubList : TSLl;

        constructor Init;
        destructor Done; virtual;
        procedure Display; virtual;
TQuadPartRec =
record
  DomX, DomY,
  RngWid,
  Sym : byte;
  S    : integer;
  D    : byte;
end;

TQuadPartRec =
^TQuadPart;
PQuadPart =
object(TPartitioner)
  CurrRange : PRangeBlockNode; { List item for current range }
  RangeList : TSLL; { Head is the current range block }
  CurrDom : PBlockNode; { Current domain block to match }
                  { with the current range block }
  DomLists : TSLL; { A list of domain lists }
  MaxDomWid, { Maximum size of a domain }
  MaxDomHt : word; { block }
constructor Init; { Descendant methods should }
{ allocate and initialize Image }
  procedure Compress(FileName : string; Tolerance : real); virtual;
    { Set up the range and domain }
    { lists and call the base }
    { compress }
private
  procedure SetUpDomainLists; virtual;
    { Create the possible domain }
    { for the various sizes of range }
    { blocks }
  procedure SetUpADomainList(Domain, Range : PRectangularBlock); virtual;
    { Create the current level of }
    { domain blocks }
  procedure WriteHeader(var f : file); virtual;
    { Write out a header that tells the }
    { image size and partitioning }
    { scheme used. }
  procedure AllocateTransforms(var CurrTrans : PTransform;
                                             var MinTrans : PTransform); virtual;
    { Allocate two of the right type }
    { of transaction objects for }
    { compress to use }
  procedure DeallocateTransforms(CurrTrans : PTransform;
                                   MinTrans : PTransform); virtual;
    { Deallocate the two working }
    { transforms }
  procedure SetUpRangeBlocks; virtual;
    { Set up the initial candidates for }
    { range blocks. }
  procedure BuildRangeBlocks(ulx,uly,wid,ht : word); virtual;
    { Quadtree recurse to build the }
    { initial range blocks }
function NextRangeBlock : PBlock; virtual;
    { Get the next available range }
    { block (Returns nil when out of )}
procedure SetUpDomainBlocks(Range : PBlock); virtual;
    { Set up the possible domain blocks }
    { for a range block }

function NextDomainBlock : PBlock; virtual;
    { Get the next domain block for the }
    { current range block (Returns nil }
    { when out of domain blocks) }

procedure WriteTransform(var f : file; Trans : PTransform); virtual;
    { Write out the necessary info to }
    { describe Trans. }

function IsDivisible(Range : PBlock) : boolean; virtual;
    { Always returns false }

procedure Split(Range : PBlock); virtual;
    { Partition a range block }

end; { TQuadPart }
PQuadTgaPart =
    TQuadTgaPart;
TQuadTgaPart =
    object(TQuadPart)
        constructor Init(TgaFile : string);
    end; { TQuadTgaPart }

destructor TPartitioner.Done;
begin { TPartitioner.Done }
    Dispose(Image,Done);
end; { TPartitioner.Done }

procedure TPartitioner.Compress(FileNamex string; Tolerance : real); var
    OutFileName : PathStr;
    OutFile : file;
    CurrRange,
    MinDom,
    CurrDom : PBlock;
    TempTrans,
    MinTrans,
    CurrTrans : PTransform;
    MaxRMS,
    MinRMS,
    RMS : TFloat;
    FoundDom : boolean;
begin { TPartitioner.Compress }
{$IFDEF logging}
    assign(logfile,'trans.out');
    rewrite(logfile);
{$ENDIF>
    OutFileName := DefaultType(FileNome,'.FIF');
    Assign(OutFile,OutFileName);
    Rewrite(OutFile,1);
    {$IFDEF logging}
    writeln(logfile,'Compressing...');
    close(logfile);
{$ENDIF>
WriteHeader(OutFile);

AllocateTransforms(CurrTrans, MinTrans);
SetUpRangeBlocks;
CurrRange := NextRangeBlock;
MaxRMS := 0;
repeat
  SetUpDomainBlocks(CurrRange); { Every range block must have at least }  
  CurrDom := NextDomainBlock; { one potential domain block }  
  FoundDom := false;
  MinRMS := NearlyInfinite;
  repeat
    RMS := CurrRange.MinimizeRMS(Image, CurrDom, CurrTrans);
    if (RMS < MinRMS)
      then begin { Found a better transform }
        MinRMS := RMS;
        MinDom := CurrDom;
        TempTrans := CurrTrans; { Swap transform pointers }
        CurrTrans := MinTrans;
        MinTrans := TempTrans;
      end;
    FoundDom := RMS <= Tolerance;
    if not FoundDom
      then CurrDom := NextDomainBlock;
  until FoundDom or (CurrDom = nil);

  if FoundDom or not IsDivisible(CurrRange)
    then begin 
      WriteTransform(DutFile, MinTrans);
      writeln('RMS error : '.MinRMS);
      writeln;
      {$IFDEF logging}
      writeln(logfile,'RMS error : '.MinRMS);
      writeln(logfile);
      {$ENDIF}
      if MinRMS > MaxRMS
        then MaxRMS := MinRMS;
    end
  else Split(CurrRange);

  CurrRange := NextRangeBlock;
until (CurrRange = nil);
DeallocateTransforms(CurrTrans, MinTrans);
Close(OutFile);
writeln('The maximum RMS error was '.MaxRMS);
end; { TPartitioner.Compress }

procedure TPartitioner.WriteHeader(var f : file);
var
  OutBuf : word;
begin { TPartitioner.WriteHeader }
  OutBuf := Image.Height;
  BlockWrite(f, OutBuf, SizeOf(OutBuf));
  OutBuf := Image.Width;
  BlockWrite(f, OutBuf, SizeOf(OutBuf));
end; { TPartitioner.WriteHeader }
procedure TPartitioner.AllocateTransforms(
  var CurrTrans : PTransform;
  var MinTrans : PTransform);
begin { TPartitioner.AllocateTransforms }
  RunError(211); { Abstract object method }
end; { TPartitioner.AllocateTransforms }

procedure TPartitioner.DeallocateTransforms(
  CurrTrans : PTransform; MinTrans : PTransform);
begin { TPartitioner.DeallocateTransforms }
  RunError(211); { Abstract object method }
end; { TPartitioner.DeallocateTransforms }

procedure TPartitioner.SetUpRangeBlocks;
begin { TPartitioner.SetUpRangeBlocks }
  RunError(211); { Abstract object method }
end; { TPartitioner.SetUpRangeBlocks }

function TPartitioner.NextRangeBlock : PBlock;
begin { TPartitioner.NextRangeBlock }
  RunError(211); { Abstract object method }
end; { TPartitioner.NextRangeBlock }

procedure TPartitioner.SetUpDomainBlocks(Range : PBlock);
begin { TPartitioner.SetUpDomainBlocks }
  RunError(211); { Abstract object method }
end; { TPartitioner.SetUpDomainBlocks }

function TPartitioner.NextDomainBlock : PBlock;
begin { TPartitioner.NextDomainBlock }
  RunError(211); { Abstract object method }
end; { TPartitioner.NextDomainBlock }

procedure TPartitioner.WriteTransform(var f : file; Trans : PTransform);
begin { TPartitioner.WriteTransform }
  RunError(211); { Abstract object method }
end; { TPartitioner.WriteTransform }

function TPartitioner.IsDivisible(Range : PBlock) : boolean;
begin { TPartitioner.IsDivisible }
  RunError(211); { Abstract object method }
end; { TPartitioner.IsDivisible }

procedure TPartitioner.Split(Range : PBlock);
begin { TPartitioner.Split }
  RunError(211); { Abstract object method }
end; { TPartitioner.Split }

{<<<<<<<<<<<<<<<<<<<<<<<<<< TSimplePart methods »»»»»»»»»»»}
procedure TSimplePart.Compress(FileName : string; Tolerance : real);
var
  RangeX, RangeY, DomX, DomY, row, col : word;
begin { TSimplePart.Compress }
  with Image do begin
    RangeX := Width div SPRWidth;
    RangeY := Height div SPRHeight;
    MaxRanges := RangeX*RangeY;
    DomX := Width div SPDWidth;
    DomY := Height div SPDHeight;
    MaxDoms := DomX*DomY;
  end;
  GetMem(RangeList,MaxRanges*SizeOf(PBlock));
  for row := 0 to RangeY-1 do for col := 0 to RangeX-1 do
    RangeList[row*RangeX+col] := New(PRectRangeBlock,
    Init(col*SPRWidth,row*SPRHeight,
    SPRWidth,SPRHeight,Image));
  GetMem(DomList,MaxDoms*SizeOf(PBlock));
  for row := 0 to DomY-1 do for col := 0 to DomX-1 do
    DomList[row*DomX+col] := New(PRectDomBlock,
    Init(col*SPDWidth,row*SPDHeight,
    SPDWidth,SPDHeight,RangeList[row]));
  TPartitioner.Compress(FileName,Tolerance);
  for row := 1 to MaxRanges do
    Dispose(RangeList[row],Done);
  FreeMem(RangeList,MaxRanges*SizeOf(PBlock));
  for row := 1 to MaxDoms do
    Dispose(DomList[row],Done);
  FreeMem(DomList,MaxDoms*SizeOf(PBlock));
end; { TSimplePart.Compress }

procedure TSimplePart.WriteHeader(var f : file);
var
  OutBuf : word;
begin { TSimplePart.WriteHeader }
  TPartitioner.WriteHeader(f);
  OutBuf := SimplePartition;
  BlockWrite(f,OutBuf,SizeOf(OutBuf));
  BlockWrite(f,SPRWidth,SizeOf(SPRWidth));
  BlockWrite(f,SPRHeight,SizeOf(SPRHeight));
  BlockWrite(f,SPDWidth,SizeOf(SPDWidth));
  BlockWrite(f,SPDHeight,SizeOf(SPDHeight));
end; { TSimplePart.WriteHeader }

procedure TSimplePart.AllocateTransforms;
var CurrTrans : PTransform;
var MinTrans : PTransform);
begin { TSimplePart.AllocateTransforms }
  CurrTrans := New(PRectTransform,Init(nil,nil,0,0,0));
  MinTrans := New(PRectTransform,Init(nil,nil,0,0,0));
end; { TSimplePart.AllocateTransforms }

procedure TSimplePart.DeallocateTransforms(
  CurrTrans : PTransform; MinTrans : PTransform);
begin { TSimplePart.DeallocateTransforms }
  Dispose(CurrTrans,Done);
  Dispose(MinTrans,Done);
end; { TSimplePart.DeallocateTransforms }

procedure TSimplePart.SetUpRangeBlocks;
begin { TSimplePart.SetUpRangeBlocks }
  RangeNum := 1;
end; { TSimplePart.SetUpRangeBlocks }

function TSimplePart.NextRangeBlock : PBlock;
begin { TSimplePart.NextRangeBlock }
  if RangeNum > MaxRanges
  then NextRangeBlock := nil
  else begin
    NextRangeBlock := RangeList'RangeNum;
    RangeNum := RangeNum+1;
  end;
end; { TSimplePart.NextRangeBlock }

procedure TSimplePart.SetUpDomainBlocks(Range : PBlock);
begin { TSimplePart.SetUpDomainBlocks }
  DomNum := 1;
end; { TSimplePart.SetUpDomainBlocks }

function TSimplePart.NextDomainBlock : PBlock;
begin { TSimplePart.NextDomainBlock }
  if DomNum > MaxDoms
  then NextDomainBlock := nil
  else begin
    NextDomainBlock := DomList'DomNum;
    DomNum := DomNum+1;
  end;
end; { TSimplePart.NextDomainBlock }

procedure TSimplePart.WriteTransform(var f : file; Trans : PTransform);
var
  OutBuf : TSimplePartRec;
begin { TSimplePart.WriteTransform }
  OutBuf.DomX := PRectangularBlock(Trans'Domain)'ulx;
  OutBuf.DomY := PRectangularBlock(Trans'Domain)'uly;
  OutBuf.S := Trunc(Trans'GrayScale*GSMult);
  OutBuf.0 := Trunc(Abs(Trans'GrayTrans));
  OutBuf.Sym := PRectTransform(Trans)'Symmetry shl 1;
  if (Trans'GrayTrans < 0)
  then OutBuf.Sym := OutBuf.Sym or 1; { Save a "sign" bit }
  BlockWrite(f,OutBuf,SizeOf(OutBuf));
  Trans.Display;
end; { TSimplePart.WriteTransform }
function TSimplePart.IsDivisible(Range : PBlock) : boolean;
begin { TSimplePart.IsDivisible }
  IsDivisible := false;
end; { TSimplePart.IsDivisible }

constructor TSimpleTgaPart.Init(TgaFile : string;
  RngWid, RngHt, DomWid, DomHt : word);
begin { TSimpleTgaPart.Init }
  if not TSimplePart.Init(RngWid, RngHt, DomWid, DomHt)
    then fail;
  Image := New(PTgalmage,Init(TgaFile));
end; { TSimpleTgaPart.Init }

constructor TBlockNode.Init(Block : PBlock);
begin { TBlockNode.Init }
  if not TSLLItem.Init
    then fail;
  BlockPtr := Block;
end; { TBlockNode.Init }

destructor TBlockNode.Done;
begin { TBlockNode.Done }
  Dispose(BlockPtr,Done);
  TSLLItem.Done;
end; { TBlockNode.Done }

procedure TBlockNode.Display;
begin { TBlockNode.Display }
  BlockPtr.Display;
end; { TBlockNode.Display }

constructor TRangeBlockNode.Init(Block : PBlock; TreeLevel : word);
begin { TRangeBlockNode.Init }
  if not TBlockNode.Init(Block)
    then fail;
  Level := TreeLevel;
end; { TRangeBlockNode.Init }

constructor TSLLItem.Init;
begin { TSLLItem.Init }
  if not TSLLItem.Init
    then fail;
    if not SubList.Init
      then fail;
end; { TSLLItem.Init }
destructor TSLLLItem.Done;
begin { TSLLLItem.Done }
    SubList.Done;
    TSLLItem.Done;
end; { TSLLLItem.Done }

procedure TSLLLItem.Display;
begin { TSLLLItem.Display }
    SubList.Display;
end; { TSLLLItem.Display }

{<<<<<<<<<<<<<<<<<<<<<<<<<<< TQuadPart methods »»»»»»»»»»»»»»}
var
row,
col,
XDoms,
YDoms : word;
CurrList : PSLLItem;
begin { TQuadPart.SetUpADomainList }
  with Image" do
  begin
    XDoms := Width div Domain".Width;
    YDoms := Height div Domain".Height;
  end;
  CurrList := New(PSLLItem,Init);
  DomLists.Append(CurrList);
  with Domain" do
  for row := 0 to YDoms-1 do
    for col := 0 to XDoms-1 do
      CurrList".SubList.Append(New(PBlockNode,
        Init(New(PRectDomBlock,
          Init(col*Width,row*Height,
          Width,Height,
          Range,Image))));

if IsDivisible(Range)
then with Range" do
  begin
    Domain".Width := Width;
    Domain".Height := Height;
    Width := Width shr 1;
    Height := Height shr 1;
    SetUpADomainList(Domain,Range);
  end; { TQuadPart.SetUpADomainList }
procedure TQuadPart.WriteHeader(var f : file);
var
  OutBuf : word;
begin { TQuadPart.WriteHeader }
  TPartitioner.WriteHeader(f);
  OutBuf := QuadTreePartition;
  BlockWrite(f,OutBuf,SizeOf(OutBuf));
end; { TQuadPart.WriteHeader }

procedure TQuadPart.AllocateTransforms(var CurrTrans : PTransform;
var MinTrans : PTransform);
begin { TQuadPart.AllocateTransforms }
  CurrTrans := New(PRectTransform,Init(nil,nil,0,0,0));
  MinTrans := New(PRectTransform,Init(nil,nil,0,0,0));
end; { TQuadPart.AllocateTransforms }

procedure TQuadPart.DeallocateTransforms(CurrTrans : PTransform;
MinTrans : PTransform);
begin { TQuadPart.DeallocateTransforms }
  Dispose(CurrTrans,Done);
  Dispose(MinTrans,Done);
procedure TQuadPart.SetUpRangeBlocks;
begin
  BuildRangeBlocks(0,0,Image^.Width,Image^.Height);
end;

procedure TQuadPart.BuildRangeBlocks(ulx,uly,wid,ht : word);
var
  HalfWid,
  HalfHt : word;
begin
  if (wid = MaxDomWid shr 1)
  then RangeList.Append(New(PRangeBlockNode,
      Init(New(PRectRangeBlock,
        Init(ulx,uly,wid,ht,Image)),
      1)))
  else begin
    HalfWid := wid shr 1;
    HalfHt := ht shr 1;
    BuildRangeBlocks(ulx,uly,HalfWid,HalfHt);
    BuildRangeBlocks(ulx+HalfWid,uly,HalfWid,HalfHt);
    BuildRangeBlocks(ulx,uly+HalfHt,HalfWid,HalfHt);
    BuildRangeBlocks(ulx+HalfWid,uly+HalfHt,HalfWid,HalfHt);
  end;
end;

function TQuadPart.NextRangeBlock : PBlock;
begin
  CurrRange := PRangeBlockNode(RangeList.Head);
  if CurrRange = nil
  then NextRangeBlock := nil
  else begin
    RangeList.Remove(CurrRange);
    NextRangeBlock := CurrRange^.BlockPtr;
  end;
end;

procedure TQuadPart.SetUpDomainBlocks(Range : PBlock);
var
  CurrDomList : PSLLList;
begin
  CurrDomList := PSLLItem(DomLists.Nth(CurrRange^.Level));
  CurrDom := PBlockNode(CurrDomList^.SubList.Head);
end;

function TQuadPart.NextDomainBlock : PBlock;
begin
  if CurrDom = nil
  then NextDomainBlock := nil
  else begin
    NextDomainBlock := CurrDom^.BlockPtr;
    CurrDom := PBlockNode(CurrDom^.Next);
  end;
end;

procedure TQuadPart.WriteTransform(var f : file; Trans : PTransform);
var
OutBuf : TQuadPartRec;
begin { TQuadPart.WriteTransform }
OutBuf.DomX := PRectangularBlock(Trans~.Domain)~.ulx;
OutBuf.DomY := PRectangularBlock(Trans~.Domain)~.uly;
OutBuf.RngWid := PRectangularBlock(Trans~.Range)~.Width;
OutBuf.S := trunc(Trans~.GrayScale*GSMult);
OutBuf.O := Trunc(Abs(Trans~.GrayTrans));
OutBuf.Sym := PRectTransform(Trans)~.Symmetry shl 1;
if (Trans~.GrayTrans < 0)
then OutBuf.Sym := OutBuf.Sym or 1; { Save a "sign" bit }
BlockWrite(f,OutBuf,SizeOf(OutBuf));

Trans~.Display; { For debugging purposes }
Dispose(CurrRange,Done); { Finished with this block }
end; { TQuadPart.WriteTransform }

function TQuadPart.IsDivisible(Range : PBlock) : boolean;
begin { TQuadPart.IsDivisible }
with PRectangularBlock(Range)~ do
IsDivisible := not (Odd(Width) or Odd(Height)) and
((Width*Height) > SizeOf(TQuadPartRec));
end; { TQuadPart.IsDivisible }

procedure TQuadPart.Split(Range : PBlock);
var
  CurrLevel,
  RngWid,
  RngHt : word;
begin { TQuadPart.Split }
with PRectangularBlock(Range)~ do
begin
  RngWid := Width shr 1;
  RngHt := Height shr 1;
  CurrLevel := CurrRange~.Level+1;

  RangeList.Insert(New(PRangeBlockNode,
    Init(New(PRectRangeBlock,
      Init(ulx+RngWid,uly+RngHt,
      RngWid,RngHt,Image)),
      CurrLevel)));
  RangeList.Insert(New(PRangeBlockNode,
    Init(New(PRectRangeBlock,
      Init(ulx,uly+RngHt,RngWid,RngHt,Image)),
      CurrLevel)));
  RangeList.Insert(New(PRangeBlockNode,
    Init(New(PRectRangeBlock,
      Init(ulx+RngWid,uly,RngWid,RngHt,Image)),
      CurrLevel)));
  RangeList.Insert(New(PRangeBlockNode,
    Init(New(PRectRangeBlock,
      Init(ulx,uly,RngWid,RngHt,Image)),
      CurrLevel)));
end;

Dispose(CurrRange,Done);
end; { TQuadPart.Split }
constructor TQuadTgaPart.Init(TgaFile : string);
begin { TQuadTgaPart.Init }
  if not TQuadPart.Init
  then fail;
  Image := New(PTgalmage,Init(TgaFile));
end; { TQuadTgaPart.Init }
end. { FractOut }

FRACTIMG.PAS

unit FractImg;

interface

uses
  FractIn,
  FractOut,
  SLList,
  Dos,
  CSUtil:

  PTransNode =
  ^TTransNode;
  TTransNode =
  object(TSLLItem)
    Range,
    Domain : PBlock;
    s,o : TFloat;
    constructor Init(Rng, Dom : PBlock; GScale, GTrans : TFloat);
    destructor Done; virtual;
    procedure Apply(var Inlmage : TImage;
                     Outlmage : PScreenView); virtual;
  end; { TTransNode }

  PRectTransNode =
  ^TRectTransNode;
  TRectTransNode =
  object(TTransNode)
    Symmetry : TSymmetry;
    XStep,
    YStep : word;
    constructor Init(Rng, Dom : PBlock; GScale, GTrans : TFloat;
                     Sym : TSymmetry);
    procedure Apply(var Inlmage : TImage;
                    Outlmage : PScreenView); virtual;
  end; { TRectTransNode }

  PFIF =
  ^TFIF;
  TFIF =
  object(TImage)
PartType : word;
Transforms : TSLL;

constructor Init(FracFile : string; Iterations : word);
destructor Done; virtual;

procedure WriteTarga(TgaFile : string);

private
  procedure ReadTransforms(var f : file); virtual;
end; { TFIF }

PSimplePartFIF =
  'TSimplePartFIF;
TSimplePartFIF =
  object(TFIF)
    constructor Init(FracFile : string; Iterations : word);
    destructor Done; virtual;

private
  procedure ReadTransforms(var f : file); virtual;
end; { TSimplePartFIF }
PQuadPartFIF =
  'TQuadPartFIF;
TQuadPartFIF =
  object(TSimplePartFIF)
    constructor Init(FracFile : string; Iterations : word);

private
  procedure QuadReadTrans(var f : file;
    x, y, wid, ht : word;
    TransRec : TQuadPartRec); virtual;
end; { TQuadPartFIF }

implementation

{<<<<<<<<<<<<<<<<<<<<<<<<<< TTransNode methods >>>>>>>>>>>>>>>>>>>>>>>>>>>}{
begin { TRectTransNode.Init }
if not TTransNode.Init(Rng,Dom,GScale,GTrans)
then fail;
Symmetry := Sym;
case Symmetry of
  0,1,2,3 : begin
    XStep := PRectangularBlock(Domain)^.Width div
     PRectangularBlock(Range)^.Width;
    YStep := PRectangularBlock(Domain)^.Height div
     PRectangularBlock(Range)^.Height;
  end;
  4,5,6,7 : begin { reflection about y=x }
    XStep := PRectangularBlock(Domain)^.Width div
     PRectangularBlock(Range)^.Height;
    YStep := PRectangularBlock(Domain)^.Height div
     PRectangularBlock(Range)^.Width;
  end;
end; { End of cases }
end; { TRectTransNode.Init }

procedure TRectTransNode.Apply(var Inlmage : TImage;
    Outlmage : PScreenView);
var
  temp : integer;
  RngHt,
  RngWid,
  row, col,
  DomRow,
  DomCol,
  RngPos : word;
begin { TRectTransNode.Apply }
  DomRow := PRectangularBlock(Domain)^.uly;
  DomCol := PRectangularBlock(Domain)^.ulx;
  RngHt := PRectangularBlock(Range)^.Height+1;
  RngWid := PRectangularBlock(Range)^.Width+1;

case Symmetry of
  0 : begin { Identity }
      for row := 0 to RngHt do
        begin
          RngPos := PRectangularBlock(Range)^.uly+
          (PRectangularBlock(Range)^.uly+row)*Inlmage.Width;
          DomCol := PRectangularBlock(Domain)^.ulx;
          for col := 0 to RngHt do
            begin
              temp :=
              Trunc(s*Inlmage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
              if temp>255 then temp := 255
              else if temp < 0 then temp := 0;
              Outlmage[RngPos+col] := temp;
              DomCol := DomCol+XStep;
            end;
          DomRow := DomRow+YStep;
        end;
  end;
end;

1: begin { reflect about the y-axis }
   for row := 0 to RngHt do begin
      RngPos := PRectangularBlock(Range).ulx +
               (PRectangularBlock(Range).uly+row)*InImage.Width;
      DomCol := PRectangularBlock(Domain).ulx;
      for col := RngWid downto 0 do begin
         temp :=
            Trunc(s*InImage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
         if temp>255 then temp := 255;
         else if temp < 0 then temp := 0;
         OutImage[RngPos+col] := temp;
         DomCol := DomCol+XStep;
      end;
      DomRow := DomRow+YStep;
   end;
end;

2: begin { reflect about the x-axis }
   for row := RngHt downto 0 do begin
      RngPos := PRectangularBlock(Range).ulx +
               (PRectangularBlock(Range).uly+row)*InImage.Width;
      DomCol := PRectangularBlock(Domain).ulx;
      for col := 0 to RngWid do begin
         temp :=
            Trunc(s*InImage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
         if temp>255 then temp := 255;
         else if temp < 0 then temp := 0;
         OutImage[RngPos+col] := temp;
         DomCol := DomCol+XStep;
      end;
      DomRow := DomRow+YStep;
   end;
end;

3: begin { 180 degree rotation }
   for row := RngHt downto 0 do begin
      RngPos := PRectangularBlock(Range).ulx +
               (PRectangularBlock(Range).uly+row)*InImage.Width;
      DomCol := PRectangularBlock(Domain).ulx;
      for col := RngWid downto 0 do begin
         temp :=
            Trunc(s*InImage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
         if temp>255 then temp := 255;
         else if temp < 0 then temp := 0;
         OutImage[RngPos+col] := temp;
         DomCol := DomCol+XStep;
      end;
end;
DomRow := DomRow+YStep;
end;
end;

4 : begin { Reflect about the line y=x }
for row := 0 to RngHt do
begin
  DomRow := PRectangularBlock(Domain).uly;
  RngPos := PRectangularBlock(Range).ulx+
            (PRectangularBlock(Range).uly+row)*Inlmage.Width;
  for col := 0 to RngWid do
  begin
    temp :=
    Trunc(s*Inlmage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
    if temp > 255
      then temp := 255
    else if temp < 0
      then temp := 0;
    Outlmage[RngPos+col] := temp;
    DomRow := DomRow+YStep;
  end;
  DomCol := DomCol+XStep;
end;
end;

5 : begin { 90 degree rotation }
for row := RngHt downto 0 do
begin
  DomRow := PRectangularBlock(Domain).uly;
  RngPos := PRectangularBlock(Range).ulx+
            (PRectangularBlock(Range).uly+row)*Inlmage.Width;
  for col := 0 to RngWid do
  begin
    temp :=
    Trunc(s*Inlmage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
    if temp > 255
      then temp := 255
    else if temp < 0
      then temp := 0;
    Outlmage[RngPos+col] := temp;
    DomRow := DomRow+YStep;
  end;
  DomCol := DomCol+XStep;
end;
end;

6 : begin { 270 degree rotation }
for row := 0 to RngHt do
begin
  DomRow := PRectangularBlock(Domain).uly;
  RngPos := PRectangularBlock(Range).ulx+
            (PRectangularBlock(Range).uly+row)*Inlmage.Width;
  for col := RngWid downto 0 do
  begin
    temp :=
    Trunc(s*Inlmage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
    if temp > 255
      then temp := 255
    else if temp < 0
      then temp := 0;
    Outlmage[RngPos+col] := temp;
DomRow := DomRow+YStep;
end;
DomCol := DomCol+XStep;
end;
end;
7 : begin { Reflect about the line y=-x }
for row := RngHt downto 0 do
begin
  DomRow := PRectangularBlock(Domain).uly;
  RngPos := PRectangularBlock(Range).ulx+
    (PRectangularBlock(Range).uly+row)*InImage.Width;
  for col := RngWid downto 0 do
  begin
    temp :=
      Trunc(s*InImage.AvgGray(DomCol,DomRow,XStep,YStep)+o);
    if temp>255
      then temp := 255
    else if temp < 0
      then temp := 0;
    OutImage[RngPos+col] := temp;
    DomRow := DomRow+YStep;
  end;
end; { End of cases }
end; { TRectTransNode.Apply }
{<<<<<<<<<<<<<<<<<<<<<<<<<<< TFIF methods »»»»»»»»»»»»»»}
constructor TFIF.Init(FracFile : string; Iterations : word);
var
  InFileName : PathStr;
  InFile : file;
  SwapScreen,
  TransScreen : PScreenView;
  count : word;
  CurrTrans : PTransNode;
begin { TFIF.Init }
  InFileName := DefaultType(FracFile,'.FIF');
  Assign(InFile,InFileName);
  Reset(InFile,1);
  BlockRead(InFile,Height,SizeOf(Height));
  BlockRead(InFile,Width,SizeOf(Width));
  BlockRead(InFile,PartType,SizeOf(PartType));
  Transforms.Init;
  ReadTransforms(InFile);
  Close(InFile);
  GetMem(MemScreen,Height*Width);
  GetMem(TransScreen,Height*Width);
  FillChar(MemScreen,Height*Width,128); { Start with an "average" screen }

  for count := 1 to Iterations do
  begin
    CurrTrans := PTransNode(Transforms.Head);
while (CurrTrans <> nil) do
begin
  CurrTrans^.Apply(Self,TransScreen);
  CurrTrans := PTransNode(CurrTrans^.next);
end;

SwapScreen := MemScreen;  
{ Swap screen chunk pointers }
MemScreen := TransScreen;
TransScreen := SwapScreen;
Display(false);
end;
FreeMem(TransScreen,Width*Height);  
{ Deallocate temp screen chunk }
end;  
{ TFIF.Init }

destructor TFIF.Done;
begin { TFIF.Done }
  Transforms.Done;
  TImage.Done;
end; { TFIF.Done }

procedure TFIF.WriteTarga(TgaFile : string);
begin { TFIF.WriteTarga }
end; { TFIF.WriteTarga }

procedure TFIF.ReadTransforms(var f : file);
begin { TFIF.ReadTransforms }
  Error('Error reading the image file','Unknown partitioning scheme');
end; { TFIF.ReadTransforms }

{<<<<<<<<<<<<<<<<<<<<<< TSimplePartFIF methods »»»»»»»»»»»»}
BlockRead(f,SPDHeight,SizeOf(SPDHeight));

RangeX := Width div SPRWidth;
RangeY := Height div SPRHeight;

for row := 0 to RangeY-1 do
  for col := 0 to RangeX-1 do
    begin
      BlockRead(f,TransRec,SizeOf(TransRec),NumRead);
      if NumRead < SizeOf(TransRec)
      then Error('Error reading transform', '');

      0 := TransRec.0;
      if (TransRec.Sym and 1) <> 0
      then 0 := -0;  { negative bit was set }

      Transforms.Insert(
        New(PRectTransNode,
          Init(New(PRectangularBlock,
            Init(Col*SPRWidth,Row*SPRHeight,
              SPRWidth,SPRHeight)),
          New(PRectangularBlock,Init(TransRec.DomX,
              TransRec.DomY,
              SPDWidth,SPDHeight)),
            (TransRec.S/GSMult),0,TransRec.Sym shr 1)));
    end;
  end;
end; { TSimplePartFIF.ReadTransforms }

constructor TQuadPartFIF.Init(FracFile : string; Iterations : word);
begin { TQuadPartFIF.Init >
  if not TSimplePartFIF.Init(FracFile,Iterations)
  then fail;
end; { TQuadPartFIF.Init }

procedure TQuadPartFIF.ReadTransforms(var f : file);
var
  TransRec : TQuadPartRec;
  DomHt,
  RngHt,
  NumRead : word;
begin { TQuadPartFIF.ReadTransforms }
  if PartType <> QuadTreePartition
  then TSimplePartFIF.ReadTransforms(f) { See if the parent can handle it }
  else begin
    BlockRead(f,TransRec,SizeOf(TransRec));
    QuadReadTrans(f,0,0,Width,Height,TransRec);
  end;
end; { TQuadPartFIF.ReadTransforms }

procedure TQuadPartFIF.QuadReadTrans(
  var f : file; x, y, wid, ht : word;
  TransRec : TQuadPartRec);
var
  D : TFloat;
  width2,
height2 : word;
begin ( TQuadPartFIF.QuadReadTrans )
if (TransRec.RngWid = wid)
then begin
  0 := TransRec.0;
  if (TransRec.Sym and 1) <> 0
  then 0 := -0;  { negative bit was set }
  Transforms.Insert(
    New(PRectTransNode,
      Init(New(PRectangularBlock,
        Init(x,y,wid,ht)),
      New(PRectangularBlock,Init(TransRec.DomX,
        TransRec.DomY,
        wid*2,ht*2)),
        (TransRec.S/GMult),0,TransRec.Sym shr 1)))
end
else begin
  width2 := wid div 2;
  height2 := ht div 2;
  QuadReadTrans(f,x,y,width2,height2,TransRec);
  BlockRead(f,TransRec,SizeOf(TransRec));
  QuadReadTrans(f,x+width2,y,width2,height2,TransRec);
  BlockRead(f,TransRec,SizeOf(TransRec));
  QuadReadTrans(f,x,y+height2,width2,height2,TransRec);
  BlockRead(f,TransRec,SizeOf(TransRec));
  QuadReadTrans(f,x+width2,y+height2,width2,height2,TransRec);
end; { TQuadPartFIF.QuadReadTrans }
end. { FractImg }
WORKS CITED

