Technical Note 05-16
画像の扱いについて

By Gerard Czwiklinski, 4D S.A.
Technical Note 05-16

（原題: Handling Pictures）

概要

画像の標準規格といっても、その種類は様々です。たとえば MacOS では PICT、Windows では BMP、インターネットでは GIF や JPEG が主流となっています。加えて、ソフトウェアに特有の形式、たとえば Photoshop の PSD や Paint Shop Pro の PSP などが存在します。4th Dimension のコマンド READ PICTURE FILE および SAVE PICTURE FILE でサポートされているのは、Quicktime が対応しているすべての画像形式です。

どんなコンピュータファイルもそうであるように、様々な画像形式といっても、その実体は、一定の規則に従って組織され、定義されたバイナリデータです。元の形式が分かっていれば、バイトレベルで内容を解読でき、さらに重要な点として、バイト単位で変換処理を施すことができます。

今回は、元の形式に左右されないように、コマンド PICTURE TO BLOB で変換してから画像を処理します。形式は、もっとも扱いが単純な BMP 形式、処理には BLOB コマンドを使用します

BMP 形式について

BMP は Windows フォーマットですが、Quicktime によって MacOS でも扱うことができるようになります。画像の種類には、白黒（1bit）、16 色（4bits）、256 色（8bits）、true color（24bits）があり、圧縮することも可能です。ここでは、true color 非圧縮を使用します。その定義は以下のとおりです。

画像には 54 ビットのヘッダが存在します。ここには画像の構造が記録されますが、今回は必要がないので説明は省略します。

1 ピクセルは 3 バイトで表現され、各バイトはピクセルの RGB 値に対応しています。

1 行のバイト数は必ず 4 の倍数でなければならず、そうでない場合は、値が 0 のデータが余分に追加されます。

たとえば、幅が 34 ピクセルの画像は、102 バイトの有効なデータと 2 バイト分の値 0 が各行のデータになります。

画像形式についての詳しい説明は下記のサイトを参照してください。

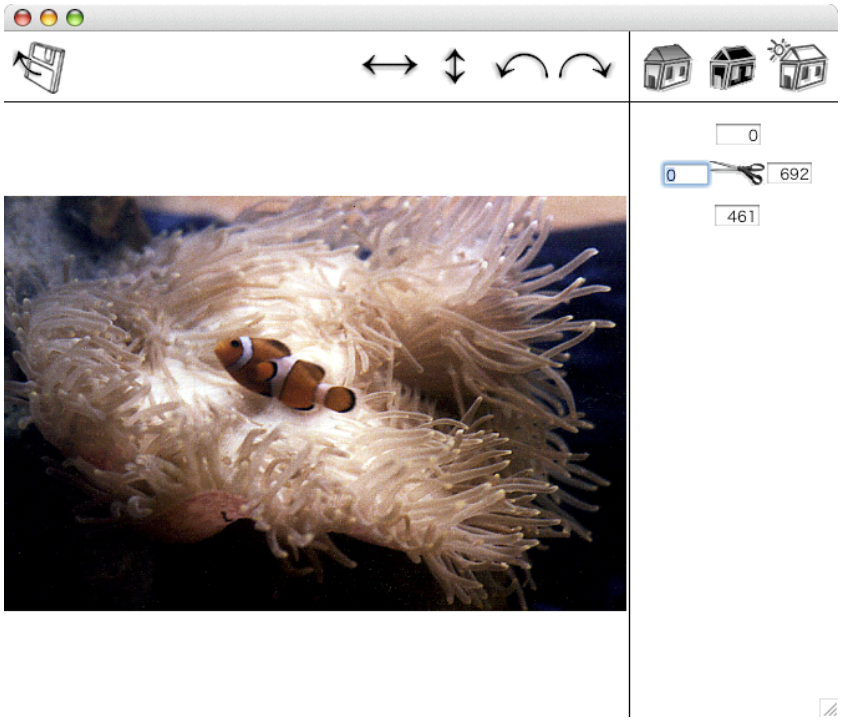
http://www.dcs.ed.ac.uk/home/mxr/gfx/2d-hi.html
http://www.wotsit.org/search.asp?s=font

サンプルデータベース

実行には Quicktime がインストール済であることことが条件になります。パフォーマンスは、コンパイルされているかによって大きく左右されるので、テストにはコンパイル版のデモ、コードの解析にはインタプリタ版のデモを使用することをお奨めします。

メイン画面

左から順に以下のボタンが用意されています。
0
画像を読み込み
横方向に反転
縦方向に反転
左に 90 度回転
右に 90 度回転
グレースケールに変換
ネガポジ反転
明度調整のダイアログを開く



上下左右の座標を選択して、はさみアイコンをクリックすれば、位置が調整できます。

左側には画像のサムネイル、右側には明るさを調整したあとのプレビューが表示されます。明るさはスライドを使用して調整します。

右下のアイコンは、明るさを元の状態に戻るために使用します。

基本原則

元の画像を 24 ビット圧縮なしの BMP 形式に変換し、BLOB に代入します。次のコマンドはその両方を一度に行なうためのものです。

```
  `1) converting picture to BMP 24 bit
  `--------------------------------------------------------------------------------
PICTURE TO BLOB($1->;$GCZ_TmpBlob;"BMPf")
```

いろいろな処理をしようとするたびに、画像の縦サイズ、横サイズが必要になってきます。これらの情報は次のコマンドで取得します。

```
  `2) Retrieving height/width -> $BitMapHeight/$BitMapWidth
  `--------------------------------------------------------------------------------
PICTURE PROPERTIES($1->;$BitMapWidth;$BitMapHeight)
```

ちなみに各行ごとのバイト数は以下の式で知ることができます。

```
  `3) Computing number of byte per line -> $NbOctetsPerLine
  `--------------------------------------------------------------------------------
  `Rq : each line is completed by 0s to match 8 bytes (32 bits)
$NbOctets:=($BitMapWidth*3)+($BitMapWidth%4)
```

このあといろいろな処理を行なうわけですが、最終的には次のコマンドで BLOB を画像に戻します。

```
  `6) Retrieving picture
  `--------------------------------------------------------------------------------
BLOB TO PICTURE($GCZ_TmpBlob;$1->)
```

まとめると以下のようになります。

```
C_POINTER($1)

C_LONGINT($res)
C_LONGINT($2;$Factor_Brightness)
```

```
$Factor_Brightness:=$2

C_LONGINT($i;$j;$k)
C_LONGINT($BitMapWidth;$BitMapHeight;$NbOctets;$NbOctetsTotal)
C_BLOB($GCZ_TmpBlob)

  `1) converting picture to BMP 24 bits
  `------------------------------------------------------------------------
PICTURE TO BLOB($1->;$GCZ_TmpBlob;"BMPf")

  `2) Retrieving height/width -> $BitMapHeight/$BitMapWidth
  `------------------------------------------------------------------------
PICTURE PROPERTIES($1->;$BitMapWidth;$BitMapHeight)

  `3) Computing number of byte per line -> $NbOctetsPerLine
  `------------------------------------------------------------------------
  `Rq : each line is completed by 0s to match 8 bytes (32 bits)
$NbOctets:=($BitMapWidth*3)+($BitMapWidth%4)

  `4) transforming bytes
  `------------------------------------------------------------------------
  `remark : for a simpler loop we process even the non displayed pixels
$NbOctetsTotal:=$NbOctets*$BitMapHeight+54
For ($i;54;$NbOctetsTotal-1;1)
        $res:=$GCZ_TmpBlob{$i}+$Factor_Brightness
        Case of
                ¥ ($res>255)
                        $res:=255
                ¥ ($res<0)
                        $res:=0
        End case

        $GCZ_TmpBlob{$i}:=$res
End for

  `5) retrieving picture
  `------------------------------------------------------------------------
BLOB TO PICTURE($GCZ_TmpBlob;$1->)
```

上下反転

もっとも簡単な操作です。1 行目と最後の行、2 行目と最後から 2 番目の行…と

いう風にデータを入れ替えて処理します。（メソッド GCZ_Img_VerticalFlip を参照。）

左右反転

行ごとに、最初のピクセルと最後のピクセル…という風にデータを入れ替えて処理します。（メソッド GCZ_Img_HorizontalFlip を参照。）

90 度回転

基本は反転と一緒で、バイトを他の位置を置き換えることによって処理します。返還後は行だったものが列、列だったものが行になります。水平、垂直幅は同一ではない場合、返還後それぞれの値は入れ替わるので、変換用の BLOB を別途用意しなければなりません。

a) 列と行を逆にします。

```
  `4a) height becomes width and vice versa
$BitMapWidth2:=$BitMapHeight
$BitMapHeight2:=$BitMapWidth
```

b)返還後の行のサイズを算出します。

隙間を埋めるためのデータ（フィラー）の存在を考慮にいれなくてはなりません。

```
  `4b) computing byte number per line
$NbOctets2:=($BitMapWidth2*3)+($BitMapWidth2%4)
```

c)変換後の BLOB サイズを算定し、BLOB 生成します。

```
  `4c) computing size and creating BLOB
$TailleDuBlob2:=$NbOctets2*$BitMapHeight2+54
SET BLOB SIZE($GCZ_TmpBlob2;$TailleDuBlob2)
```

ヘッダはほとんど元と変わりませんが、幅と高さの情報（オフセット 18 と 24）だけが逆になります。

d)ヘッダを生成します。

```
  `4d)  Creating header
```

```
  `through copy ...
COPY BLOB($GCZ_TmpBlob;$GCZ_TmpBlob2;0;0;54)
  `... and modifying header of original picture
$offset:=18
LONGINT TO BLOB($BitMapWidth2;$GCZ_TmpBlob2;PC byte ordering ;$offset)
LONGINT TO BLOB($BitMapHeight2;$GCZ_TmpBlob2;PC byte ordering ;$offset)
```

注: BMP は元来 Windows のフォーマットなので、pc Byte Ordering を使用します。

（メソッド GCZ_Img_RightRotation、GCZ_Img_LeftRotation を参照。）

トリミング

変換後の画像寸法は、元とは異なるので、別の BLOB を用意して特定のピクセルだけを対象 BLOB にコピーします。(メソッド GCZ_Img_CropImage を参照。)

グレースケールに変換

グレースケール画像の場合、RGB 各要素の値はどれも同じになりますが、その値は次の式によって算出できます。

Red=Green=Blue= (Orig_Red+Orig_Green+Orig_Blue)/3

人間の目は光の 3 原色に対して均一に反応するわけではない点を考慮し、次のように修正することができます。

Red=Green=Blue=(0,3*Red)+(0,6*Green)+(0,1*Blue)

（メソッド GCZ_ImgFlt_GrayScale を参照。）

ネガポジ反転

このアルゴリズムは簡単です。すべてのピクセルに対して次の処理を実行するだけです

```
Red:=255-Red
Green:=255-Green
Blue:=255-Blue
```

（メソッド GCZ_ImgFlt_Negative を参照。）

明るさの調整

Red:=Red + input
Green:=Green+ input
Blue:=Blue+ input

（メソッド GCZ_ImgFlt_Brightness を参照。）

プログラム上の注意点

のように、いくつもの演算を実行する場合、データタイプを正しく宣言しておくことは極めて重要です。以下はコンパイルしたデータベースのテスト結果です。

（テストに使用したメソッド：ネガポジ反転）

```
$offset:=54
For ($i;1;$BitMapHeight)
	For ($j;1;$BitMapWidth*3)
		$GCZ_TmpBlob{$offset}:=255-$GCZ_TmpBlob{$offset}
		$offset:=$offset+1
	End for
	$offset:=$offset+$NbOctetsFiller
End for
```

実験 1　コンパイラ宣言を 4D に任せる場合（デフォルトで実数型）

処理に要した時間= 2340 ms

実験 2　ループ変数の型（$i）を宣言

　コンパイラは自動的にループを最適化するので、変数のタイプによるテスト結果の違いはほとんどありません。

実験 3　オフセット変数（$offset）の型を宣言

明示的に倍長整数型で宣言するだけでパフォーマンスが飛躍的に（約 8 倍）向上します。

処理に要した時間= 307ms

実験 4　演算に関わる変数（$NbOctetsFiller）の型を宣言

演算処理を行なう変数同士（この場合は$i と$NbOctetsFiller）の型を合わせることは極めて重要です。仮に一方が整数で他方が実数だった場合、4D はまずデータを変換してから実際の演算にとりかかることになります。

処理に要した時間= 260ms

メモリ管理

JPEG などの画像形式は、データを圧縮して管理しています。これを 24 ビット圧縮なしの BMP などに変換すれば、かなりのメモリを消費することになりかねません。

たとえば、1024x768、40%圧縮、サイズ 44kb の JPEG 画像は、ビットマップだと 2.25Mb（50 倍の容量）になります。

今回のサンプルを、メモリ消費という観点から改めて考えてみましょう。

READ PICTURE FILE(document;GCZ_Picture)

この時点では RAM メモリの領域 44kb を占有しています。

PICTURE TO BLOB($1->;$GCZ_TmpBlob;"BMPf")

元の画像は参照で渡されているので複製されず、44kb のままですが、結果 BLOB のサイズは 2.25Mb になります。

```
COPY  BLOB($GCZ_TmpBlob;$GCZ_TmpBlob2;0;0;
BLOB Size($GCZ_TmpBlob))
```

回転系のメソッドは 2 個目の BLOB が必要です。元の画像 44kb、BLOB 画像 2.25x2 個の計 5.5Mb がいまや画像に占有されている状態です。

BLOB TO PISTURE($GCZ_TmpBlob;$1->)

処理を終え、BLOB を画像に返すと、44kb だったサイズは 2.25Mb になります。処置の過程で使用した BLOB が 2.25Mb、回転ならばもうひとつ 2.25Mb、最悪 6.75Mb までのメモリを使用することになってしまいます。

BLOB はローカル変数なのでメソッド終了とともに消滅しますが、ピクチャの容量が巨大化したことにはかわりなく、これは表示速度その他 4D のパフォーマンスに深刻な影響を与えることになりかねません。

これを回避するには以下のような方法が考えられます。

方法 1

4D に割り当てるメモリを殖やすことによる方法。4D 2004 では特に必要ありませんが、Windows および OS 9.2 の 4D 2003 では、割り当てるメモリの量を設定する必要があります。

方法 2

画像はディスク上で扱うことによる方法。BMP ファイルを処理するようにメソッドを書き換えます。

方法 3

結果の画像を再変換することによる方法。フォーマットやサイズを変更します。JPEG に変換する場合は、以下のようなコードになります。

```
`Creation of the BMP picture
BLOB TO PICTURE(blob;image)
`Conversion into JPEG
PICTURE TO BLOB (image;blob;"JPEG")
BLOB TO PICTURE(blob;image)
```

次のコマンドも画像を変換することができます。

COMPRESS PICTURE、PICTURE TO GIF、SAVE PICTURE TO FILE