Contents

1 Introduction

This is a manual for the installation and use of the application ‘PF4File’. The program is a Windows application, for creating and viewing a .pf4 file. This is an emerging file format which currently uses up to 256 distinct colours. With this program one can perform two functions:

1. Create a .pf4 static image file from a .bmp file and conversely.
2. Create a .pf4 video file from a series of .bmp files and conversely.

The pf4 specification is a lossless compression which results in file sizes of 1/6 to 1/15 (or more) of the equivalent bitmap image.

The .pf4 specification is published in this manual for both static images and video files. This application permits the user to gain familiarity with the format and its advantages.

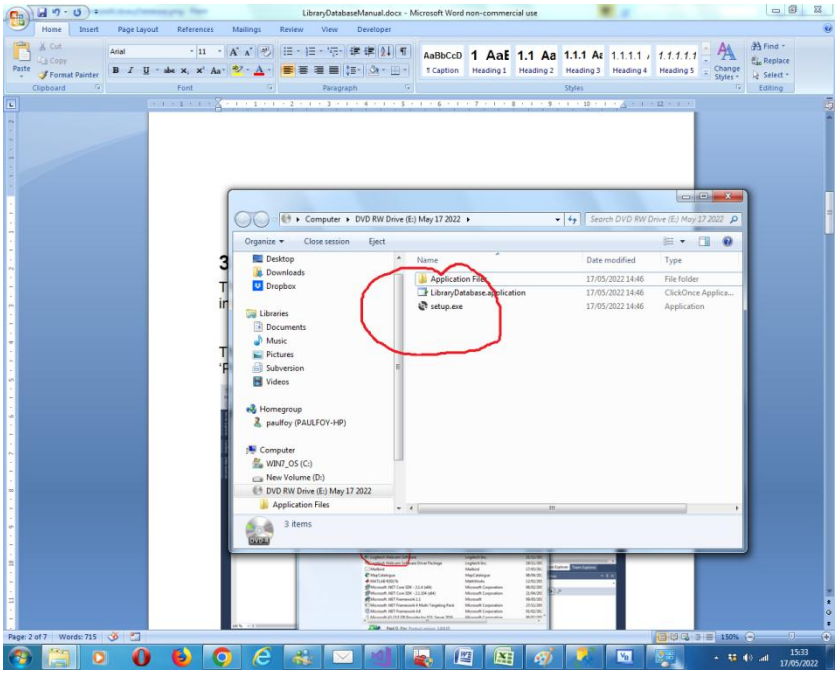
2 Pre-requisites.

1. A PC running Windows 7 or above.

2. A USB stick or optical drive containing the program setup files, together with this manual (available online).

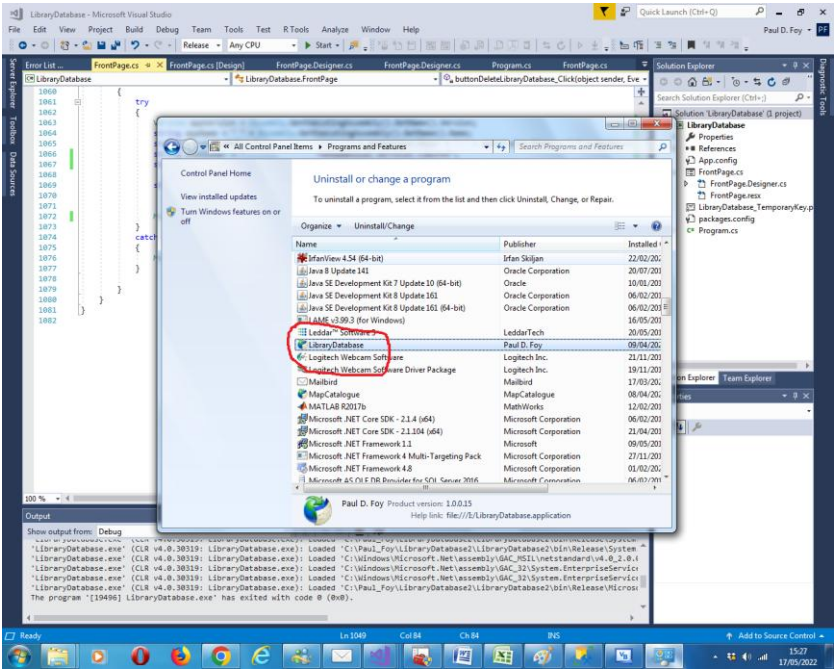
3 Installation/Removal

The program is installed by inserting the supplied stick or disc into the PC and running the ‘setup.exe’ program on it.



The program can be removed from the PC, by using the ‘Program & Features’ menu from within Control Panel.

The program is robust to erroneous parameter input, but not completely fool proof.



4 Current Limitations

There are a few limitations to the program as it currently stands (March 2025):

1. The program is limited to files and videos with 256 colours or less. This will be checked prior to creating an image or video.
2. The ‘Player’ of the .pf4 video loaded is not a professional one and there is no streaming of video frames whilst playing, for example. All frames must have been loaded (done by the application) prior to play and this can take some time for a large video. There must be plenty available RAM on the computer running the application in this case.

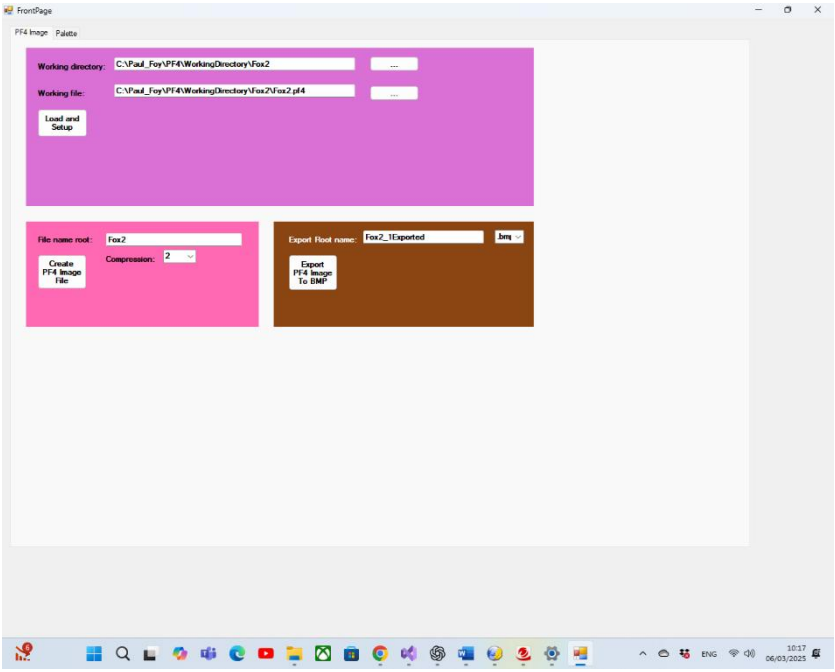
It is recommended to run the application (particularly when creating or replaying videos) on a PC with a reasonable amount of memory (RAM and hard disc space).

5 PF4File

The application has currently four tabs, two for those of a static image and two for those of a video:

5.1 Static Image

5.1.1 PF4Image



5.1.1.1 Top panel

The text box **Working directory** is the directory where files created will be saved. This must be selected before the application will function.

The text box **Working file** is the file that the newly created .pf4 file is based upon.

The button **Load and Setup** loads either the .pf4 or the .bmp file into memory and displays it in the Palette.

5.1.1.2 Bottom left panel

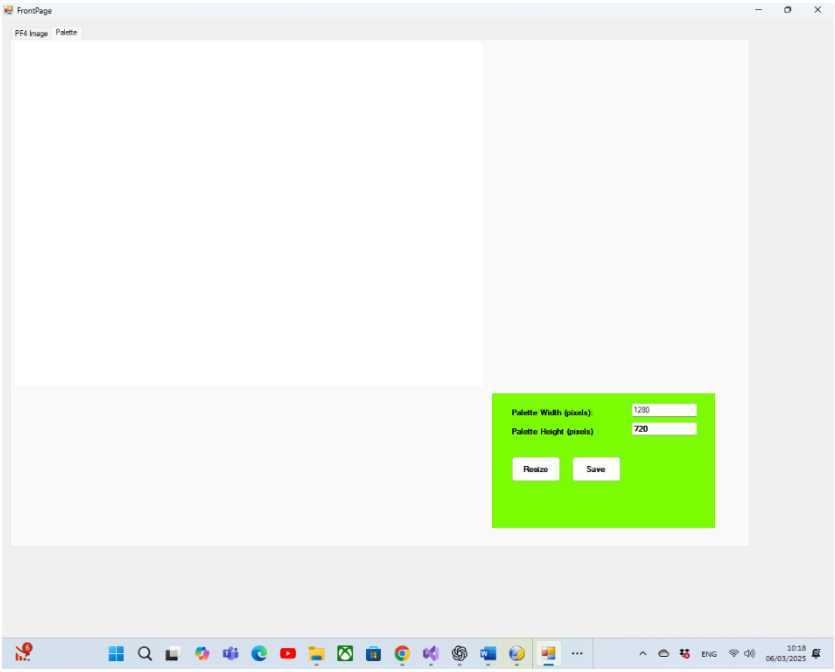
A .pf4 file with the root name of **File name root** is created with compression of **Compression** and saved in the **Working directory**. This is the function of the button **Create PF4 Image File**. Compression c reduces the width and height of the image by a factor of $1/c$ on saving ($1 \leq c \leq 15$). The width and height must be a multiple of c.

The algorithm for Compression and Decompression will be described in due course. It is a lossless one.

5.1.1.3 Bottom right panel

The button **Export PF4 Image To BMP** exports the .pf4 file created to a .bmp and saves it in the **Working directory**. **Export root name** is the filename root.

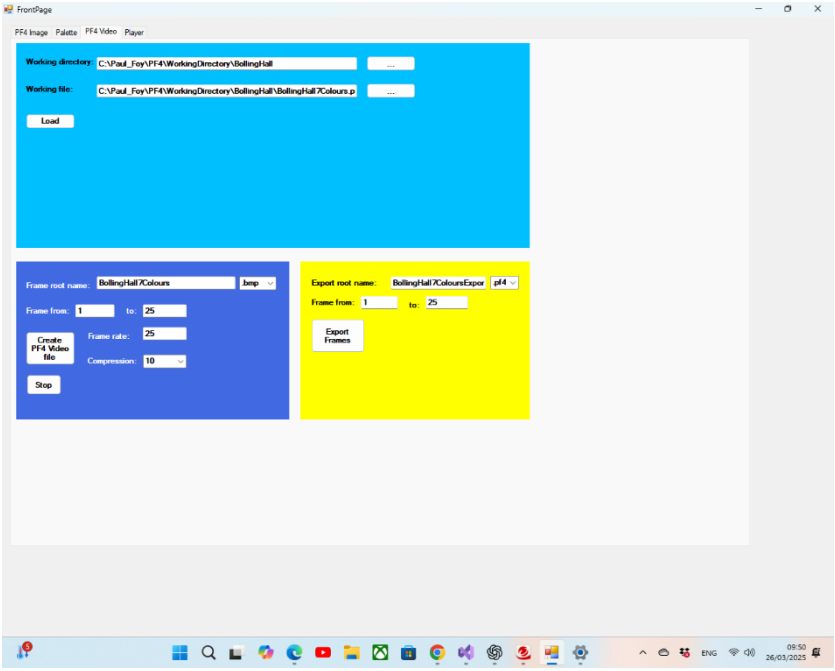
5.1.2 Pallet



This tab contains a picture box control for receiving the created image. The image may be resized with a new or original size by using the **Palette Width (pixels)**, **Palette Height (pixels)** text boxes and **Resize** button. The image is saved with the **Save** button.

5.2 Video

5.2.1 PF4Video



5.2.1.1 Top Panel

The **Working directory** for videos can be selected. This is where all created videos and exported frames can be found. In the **Working file** textbox a .pf4 file can be selected and then loaded. The **Load** button performs this load. Not that for a large .pf4 file this may take some time.

5.2.1.2 Bottom Left Panel

The **Create PF4 Video file** creates a .pf4 video file from a series of .bmp frames from **Frame from** to frame **to**. The

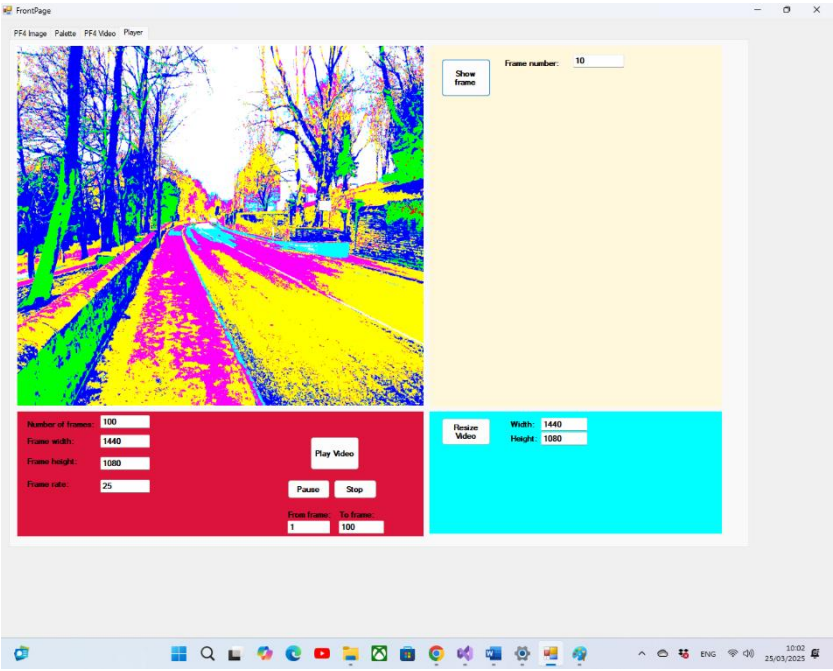
names of the bmp files must be of the form Frame root name plus ‘_1’, ‘_2’, … etc. **Frame rate** is the frame rate in frames per second and **Compression** is the compression employed for individual frames of the video. The algorithm for compression of individual frames of the video is as described in section 6. Be patient as the creation for large videos may take some time.

As the creation can take some time the process can be irretrievably stopped by using the **Stop** button.

5.2.1.3 Bottom Right Panel

The individual frames of a .pf4 video may be created to disc via the **Export Frames** button. The exported filetype may be either .pf4 or .bmp. The frames to export are **Frame from** to frame **to**. The text box **Export root name** gives the root name of the exported frames.

5.2.2 Player



5.2.2.1 Bottom Left Panel

A loaded video may be played via the **Play Video** button and can be paused and stopped with the **Pause** and **Stop** button respectively. The video is played **From frame** to **To frame**. The **Number of frames** in the video, the **Frame Width**, **the Frame Height** and the **Frame rate** are displayed.

5.2.2.2 Bottom Right Panel

The picture box control for display of video and frames may be resized to **Width** and **Height**.

5.2.2.3 To Right Panel

The button **Show frame** display the **Frame number** indicated.

6 File Byte Structure of a .pf4 File

This section describes the file byte structure of a .pf4 file as created by this application. It also permits the creation of a .pf4 file by an independent program which can then be loaded by this application, or an independent program.

A file '`inputfile`' would be read into an array of bytes by a line of C# code such as in Table 1

```
byte[] pf4Data = File.ReadAllBytes(inputfile);
```

Table 1

Similarly an array of bytes would by written to file, '`filename`' by a line of C# code such as in Table 2.

```
File.WriteAllBytes(filename, byteArray);
```

Table 2

`byteArray` is a sequence of bytes having the following structure (and in the order stated):

[FileType][NumberOfColours][Compression][Colours][ImageWidth][ImageHeight][SpatialTrackLengths][SpatialTracksDuration][DictionarySize][DictionaryBytes][SpatialTrackColours]

For a static image there is a single instance of [SpatialTrackLengths][SpatialTracksDuration][DictionarySize][DictionaryBytes][SpatialTrackColours]. For a video the fields [SpatialTrackLengths][SpatialTracksDuration][SpatialTrackColours] repeat for each frame of the video.

[FileType] – 1 byte, set to 0 (to indicate a static image as opposed to a 1 for a video).

[NumberOfColours] – 1 byte, set to the number of distinct colours in the image.

[Compression] – 1 byte, set to the value of c, as described in section 5.1.1.2.

[Colours] – 3 x NumberOfColours bytes. For each colour 3 bytes are supplied representing the red, green and blue values (in that order consecutively and one colour followed by another).

[ImageWidth] – 4 bytes, an integer representing the width (in pixels) *of the compressed image*.

[ImageHeight] – 4 bytes, an integer representing the height (in pixels) *of the compressed image*.

The next four fields are used to describe how the structure of the compressed image is described.

If w and h are the width and height of the original image, then $\bar{w} = w/c$ and $\bar{h} = h/c$ are the width and height of the compressed image respectively. $\bar{w}\bar{h}$ SpatialTracks are saved each having SpatialTrackLength, where each SpatialTrack consists of up to c^2 *trails* as described subsequently. Values for c of 8 and 10 are recommended.

Colours, of a single pixel, are stored by their *index number* (corresponding to a number between 0 and 255) as in their location in the [Colours] array, and not by RGB values.

SpatialTrackLengths and SpatialTrackDuration and SpatialTrackColours will firstly be described, by example, by referring to the 16 pixels corresponding to the case where $c = 4$, and to the 25 pixels in the case where $c = 5$.

1	2	5	10
3	4	7	12
6	8	9	14
11	13	15	16

Table 3

1	2	5	10	17
3	4	7	12	19
6	8	9	14	21
11	13	15	16	23
18	20	22	24	25

Table 4

The 16 or 25 pixels are cycled through in the order of the numbering. Trails are tuples representing the number of occurrences of the colour and the colour index in that order. The pixels are cycled through until they are exhausted. They may be up to 16 or 25 trails. The idea is that the ordering of Table 3 and Table 4, is most likely to achieve an efficiency in the storage because similar colours are most likely to be found corresponding to this sequence, as the ordering is in ascending order of the distance from the top left index..

Thus the trails and the byte storage for a cell with the (index of) colours of Table 3 is:

0	0	12	15
0	0	12	13
12	12	15	14
15	13	14	14

Table 5 (Colours with index values 12, 13, 14,15)

(4,0),(4,12),(3,15),(2,13),(3,14)

Figure 1

Note that the sum of the values for the first ordinate is 16. Thus we have:

[SpatialTrackLengths] – This is a byte representing the number of bytes in all the trails of SpatialTrackColours or SpatialTrackDuration. Thus in Figure 1 it is 10. There is an entry for each of the $\bar{w}\bar{h}$ spatial tracks and the order is from left to right top to bottom.

[SpatialTrackDuration] – these are bytes representing the length of each trail. Thus it is the sequence 4,4,3,2,3 in Figure 1. The ordering of the sequences is left to right top to bottom.

[DictionarySize] – This is a single byte that gives the number of bits that are needed to store NumberOfColours. Thus it is the smallest integer d such that $2^d \geq NumberOfColours$.

[DictionaryBytes] – Each colour is represented by d bits. This is the dictionary of colours. Each colour's representation is saved

as a byte. The order of the bytes is the order in which the colours were enumerated.

[SpatialTrackColours] – this is a sequence of bytes representing the spatial track index colour of each trail. Thus it is the sequence 0,12,15,13,14 in Figure 1. There is a sequence for each of the $\bar{w}\bar{h}$ spatial tracks and the order is from left to right top to bottom. The bytes are not saved directly but the indices of the colours are saved and then retrieved as bits according to their dictionary representation.

6.1 Code Fragmets

Some code fragments are given to assist in writing code both to create a .pf4 file and to read in from one.

An example of writing to a byte array to populate the fields, to save to disc, would be as in Figure 2. Note that if we are writing a video the code in blue is repeated for each frame of the video.

```
List<byte> byteArray = new List<byte>();

byteArray.Add(FileType);
byteArray.Add(NumberOfColours);
byteArray.Add(Compression);

for (int i = 0; i < NumberOfColours; i++)
{
    byteArray.Add(Colours[3 * i]);
    byteArray.Add(Colours[3 * i + 1]);
    byteArray.Add(Colours[3 * i + 2]);
}

byte[] bytesInInt = BitConverter.GetBytes(ImageWidth);
for (int i = 0; i < bytesInInt.Length; i++)
{
    byteArray.Add(bytesInInt[i]);
}
```

```
bytesInInt = BitConverter.GetBytes(ImageHeight);
for (int i = 0; i < bytesInInt.Length; i++)
{
    byteArray.Add(bytesInInt[i]);
}

if (FileType == 0)
{

    List<bool>[] dictionary = new
List<bool>[NumberOfColours];
byte[] dictionaryByte = new byte[NumberOfColours];
byte dictionarySize = 2;
//Create dictionary
createDictionary(ref dictionary, ref dictionarySize);
writeDictionaryParameters(dictionary, dictionarySize,
ref byteArray);

    for (int n = 0; n < ImageHeight; n++)
    {
        for (int m = 0; m < ImageWidth; m++)
        {
            byteArray.Add(SpatialTrackLengths[m, n]);
        }
    }

    for (int n = 0; n < ImageHeight; n++)
    {
        for (int m = 0; m < ImageWidth; m++)
        {
            for (int l = 0; l < SpatialTrackLengths[m,
n]; l++)
            {
                byteArray.Add(SpatialTracksDuration[m,
n][l]);
            }
        }
    }

    setSpatialTrackColours(ref byteArray);
```

```
}
public void createDictionary(ref List<bool>[]
dictionary, ref byte dictionarySize)
 {
     //Find how many bits we need to represent all the
colours.
     dictionarySize = 2;
     for (int i = 0; i < 8; i++)
     {
         if (Math.Pow(2, i) >= NumberOfColours)
         {
             dictionarySize = (byte)i;
             break;
         }
     }


     List<List<bool>> listToSelectFrom =
returnLists(dictionarySize);



     //populate the dictionary
     for (int l = 0; l < NumberOfColours; l++)
     {
         dictionary[l] = listToSelectFrom[l];
     }
 }
```

```
 public void writeDictionaryParameters(List<bool>[]
dictionary, byte dictionarySize, ref List<byte>
byteArray)
 {
     byte[] dictionaryByte = new byte[NumberOfColours];


     for (int i = 0; i < NumberOfColours; i++)
     {
         convertToByte(dictionary[i], ref
dictionaryByte[i]);
     }


     byteArray.Add(dictionarySize);


     for (int k = 0; k < NumberOfColours; k++)
     {
         byteArray.Add(dictionaryByte[k]);
     }
 }
private void setSpatialTrackColours(ref List<byte> byteArray)
{
   List<bool> ST = new List<bool>();
   List<bool>[] dictionary = new List<bool>[NumberOfColours];
   byte[] dictionaryByte = new byte[NumberOfColours];
   byte dictionarySize = 2;
   //Create dictionary
   createDictionary(ref dictionary, ref dictionarySize);
```

```
    for (int i = 0; i < NumberOfColours; i++)
    {
        convertToByte(dictionary[i], ref dictionaryByte[i]);
    }

    byteArray.Add(dictionarySize);

    for (int k = 0; k < NumberOfColours; k++)
    {
        byteArray.Add(dictionaryByte[k]);
    }

    for (int n = 0; n < ImageHeight; n++)
    {
        for (int m = 0; m < ImageWidth; m++)
        {
            for (int l = 0; l < SpatialTrackLengths[m, n]; l++)
            {
                for (int k = 0; k < dictionary[SpatialTracksColours[m,
n][l]].Count; k++)
                {
                    ST.Add(dictionary[SpatialTracksColours[m, n][l]][k]);
                }

            }
        }
```

```
    }
    BitArray STArray = new BitArray(ST.Count);
    for (int i = 0; i < ST.Count; i++)
    {
        STArray[i] = ST[i];
    }
    byte[] STByteArray = new byte[1 + STArray.Count / 8];
    STArray.CopyTo(STByteArray, 0);
    for (int l = 0; l < STByteArray.Length; l++)
    {
        byteArray.Add(STByteArray[l]);
    }
}
```

Figure 2

An example of reading the fields of the .pf4 file ready for use in an application would be as in Figure 3. If we are reading from a video the code highlighted in blue has got to be repeated for each frame of the video.

```
//Create PF4 object from disk file.
int iCount = 0;  //This is the number of bytes read.
byte[] pf4Data = File.ReadAllBytes(inputfile);
FileType = pf4Data[iCount];
iCount += 1;
NumberOfColours = pf4Data[iCount];
iCount += 1;
Compression = pf4Data[iCount];
iCount += 1;
//3 bytes per colour
Colours = new byte[3 * NumberOfColours];
for (int i = 0; i < NumberOfColours; i++)
{
    Colours[3 * i] = pf4Data[iCount + 3 * i];
    Colours[3 * i + 1] = pf4Data[iCount + 3 * i + 1];
    Colours[3 * i + 2] = pf4Data[iCount + 3 * i + 2];
}
iCount += (3 * NumberOfColours);
//2 bytes for the width
byte[] bytesInInt = new byte[sizeof(int)];
for (int i = 0; i < sizeof(int); i++)
{
```

```
    bytesInInt[i] = pf4Data[iCount + i];
}
ImageWidth = BitConverter.ToInt32(bytesInInt, 0);
iCount += sizeof(int);
//2 bytes for the height
int imageWidth = ImageWidth;
bytesInInt = new byte[sizeof(int)];
for (int i = 0; i < sizeof(int); i++)
{
    bytesInInt[i] = pf4Data[iCount + i];
}
ImageHeight = BitConverter.ToInt32(bytesInInt,0);
int imageHeight = ImageHeight;
iCount += sizeof(int);
if (FileType == 0)
{
    //we have a static image

    //The spatial track lengths
    SpatialTrackLengths = new byte[imageWidth, imageHeight];
    int totalSpatialTrackLength = 0;
    for (int n = 0; n < imageHeight; n++)
    {
        for (int m = 0; m < imageWidth; m++)
        {
            SpatialTrackLengths[m, n] = pf4Data[iCount + (n * imageWidth
```

```
+ m)];

            totalSpatialTrackLength += SpatialTrackLengths[m, n];
        }
    }
    iCount += (imageWidth * imageHeight);

    //the spatial tracks themselves
//The dictionary
   DictionarySize = 2;
   Dictionary = new List<bool>[NumberOfColours];
   getDictionary(ref pf4Data, ref DictionarySize, ref Dictionary, ref
iCount);
    SpatialTracksDuration = new List<byte>[imageWidth, imageHeight];
    SpatialTracksColours = new List<byte>[imageWidth, imageHeight];
    int iSpatialTrackCount = 0;
    for (int n = 0; n < imageHeight; n++)
    {
        for (int m = 0; m < imageWidth; m++)
        {
            SpatialTracksDuration[m, n] = new List<byte>();
            for (int l = 0; l < (SpatialTrackLengths[m, n]); l++)
            {
                SpatialTracksDuration[m, n].Add(pf4Data[iCount + l]);
            }
            iCount += (SpatialTrackLengths[m, n]);
```

```
            iSpatialTrackCount += (SpatialTracksDuration[m, n].Count);
        }
    }

    getSpatialTrackColours(ref pf4Data, iCount);
}
private void getSpatialTrackColours(ref byte[] pf4Data, ref int iCount,
byte dictionarySize, List<bool>[] dictionary)
{

    byte[] byteArray = new byte[pf4Data.Length - iCount];
    for (int i = 0; i < byteArray.Length; i++)
    {
        byteArray[i] = pf4Data[iCount + i];
    }
    BitArray STColours = new BitArray(byteArray);
    int iCount_ = 0;

    for (int n = 0; n < ImageHeight; n++)
    {
        for (int m = 0; m < ImageWidth; m++)
        {
            SpatialTracksColours[m, n] = new List<byte>();

            for (int l = 0; l < (SpatialTrackLengths[m, n]); l++)
            {
```

```
            reconstructSpatialTracks(iCount_, m, n, dictionarySize,
STColours, dictionary);
            iCount_ += dictionarySize;


         }



      }
   }

   if (((double)iCount_ / 8) == (iCount_ / 8))
   {
      iCount += (iCount_ / 8);
   }
   else
   {
      iCount += (1 + iCount_ / 8);
   }

}
```

Figure 3

Paul D. Foy
Mathematical Services

March 2025