Implementing a GIF Decoder

Decoding and displaying GIF images using C++

Andrew S. Downs
andrew@downs.ws

Abstract

Many client-side applications need to display images, and the GIF format remains popular despite its age. Various libraries exist that can decode GIFs, but sometimes for custom applications you may need a smaller memory footprint or better control over the process. In this paper I discuss the decoding algorithm, and the design and implementation in C++ of a GIF module that can plug-in to a larger image handling architecture.

Introduction

GIF (Graphics Interchange Format) uses the LZW (Lempel-Ziv-Welsh) compression scheme. The encoding and decoding processes both build a dictionary of codes representing unique RGB color sequences within the image data. The dictionary is not explicitly included with the image data. An encoder generates the dictionary while reading the uncompressed data, and a decoder regenerates the dictionary as it reads codes from the compressed stream. As the parser inspects the data stream, when it encounters a new sequence it writes the next code to the dictionary and enters the appropriate color information. If using a tree structure to hold the codes, which may be desirable in order to save space, the parser also needs to save a reference to the parent code.

Why bother writing a decoder when libraries exist that handle this and more? One reason is that many libraries are large. You may not want to ship or load something that contains a lot of unused functionality for your application. Or you may want to bypass a particular decoder implementation and substitute something faster or smaller. Or perhaps you only need to display GIFs but the target system is known to have little memory or storage space (perhaps an embedded system). Whatever the reason, you may find a need to implement a custom GIF decoder.

File format

The GIF file format is flexible and allows multiple images as well as extensions that may be application-specific. But the core structure is simple. A GIF file begins with a fixed-size area containing a header (with GIF version identifier), screen descriptor, and global color table data. The image data follows the color table.

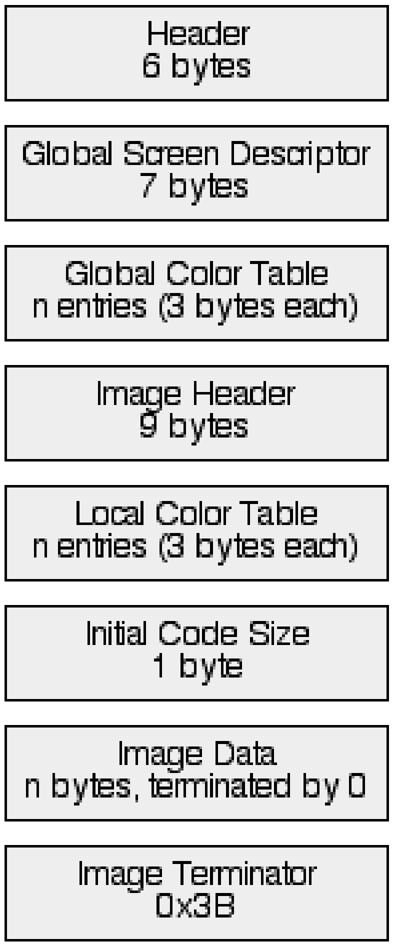

Figure 1. Image file structure.

Header

The image type is identified by the first six bytes in the file, either "GIF87a" or "GIF89b". Here is how this might be declared as a structure in C/C++;

```
struct Header {
  Byte signature[3];
  Byte version[3];
} ;
```

Note that the data type `Byte` is defined as:

```
#define Byte unsigned char
```

The type Byte has been used for many years in the Mac OS Toolbox, but it is not part of the ANSI C standard. This macro equates Byte with an unsigned character, defined in ANSI C as an 8-bit type. Byte thus provides a shorthand way to reference the unsigned char data type in this application.

Following that identifier is the Logical Screen Descriptor, which contains information about the dimensions of the overall area occupied by the images in the file (they may occupy different parts of the screen), various flags, and color information.

```
struct LogicalScreenDescriptor {
  Byte logicalScreenWidth[ 2 ];
  Byte logicalScreenHeight[2];
  Byte bitField;
  Byte backgroundColor;
  Byte pixelAspectRatio;
} ;
```

Colors are specified in a color table, up to a maximum of 256 colors. Images may each define separate tables or share the global table. The code discussed in this paper assumes a global color table. Colors are all in a 3-byte RGB format as shown here.

```
struct ColorTableEntry {
  Byte red;
  Byte green;
  Byte blue;
} ;
```

GIF87a defines two specific identifiers that mark the sections following the global color table. The start of an image is marked by 0x2c, and the end of the image by 0x3b. In-between is the compressed data stream.

```
const Byte gTypeImageBlock = 0x2c;
const Byte gTypeTerminator = 0x3b;
```

GIF89b added extension blocks, which allow for text overlays, animation frames, and generic and application-specific data. Each block type is identified at the start of the block by 0x21 plus a type-specific value. These extensions are handled in the accompanying code, but nothing is done with the contents.

```
const Byte gTypeExtension = 0x21;
```

An image header precedes each image in the file. This defines the x-y coordinates for the upper-left corner of the image and the image width and height. A single byte specifies whether or not a local color table is used (and its size), and whether or not the image data is interlaced.

```
struct ImageHeader {
  Byte leftPosition[2];
  Byte rightPosition[2];
  Byte imageWidth[2];
  Byte imageHeight[2];
  Byte bitField;
} ;
```

The code dictionary

The codes are stored in a physical array, but the logical structure is a tree, and each node can trace its lineage through exactly one parent up to a root node. A code is really a unique sequence of tree entries. Each tree entry contains individual RGB values and the parent's index. Here is the structure for an entry:

```
  struct DictionaryTreeEntry {
    Byte red;
    Byte green;
    Byte blue;
    unsigned int parent;
  } ;
```

The array is declared as:

```
struct DictionaryTreeEntry dictionaryTree[ 4096 ];
```

The dictionary gets initialized using the color values from the color table (in this example, the global color table). For a code size of 8-bits, the first 256 values in the dictionary are the color table entries. See listing 1 for an example of setting up the dictionary.

```
void GifUtils::InitializeDictionary( int size ) {
  // size determines the number of color table entries
  int limit = ( int )pow( 2, size );

  clearcode = ( int )pow( 2, size );
  endcode = clearcode + 1;
  startcode = clearcode + 2;
  nextcode = startcode;

  currCodeSize = size + 1;

  for ( int i = 0; i < limit; i++ ) {
    // Copy the appropriate color table source byte to its
    // dictionary counterpart.
    dictionaryTree[ i ].blue =
      *( ( globalColorTable + ( i * 3 ) + 2 ) );

    dictionaryTree[ i ].green =
      *( ( globalColorTable + ( i * 3 ) + 1 ) );

    dictionaryTree[ i ].red =
      *( ( globalColorTable + ( i * 3 ) ) );

    // Color table entries have no parent.
    dictionaryTree[ i ].parent = 0;
  }

  stackPointer = 0;
}
```

Listing 1. Initializing the dictionary using the color table.

Image decoding

There are several steps involved in decoding: read the next code value, determine how it fits into the dictionary, and write an RGB sequence when appropriate.

The image data in LZW consists of a series of codes that are used to build a dictionary. Each code entry or node in the dictionary maps to an RGB value. Each node other than the color table entries also has a parent node. The color table entries are root nodes. The RGB stream associated with a particular node consists not only of its RGB values, but also those of each parent up to and including the root node.

The first method, shown in listing 2, handles one of the more tricky parts of the LZW algorithm. The number of bits used to encode the sequences is not constant. It starts as the value specified in the Initial Code Size byte, which follows the Image Header. As more codes are added to the dictionary, the number of available codes gradually dwindles to zero, at which point the number of bits used to represent a code requires incrementing. Reading and dictionary building continues until the number of available codes is once again exhausted, at which point the size increments again. For small images without much color variation this may not be much of an issue, but the decoder must be prepared to handle it. The tricky part in reading variable-length values from the data stream involves remembering what portion of the last byte was left unused, how many bits need to be read to make up the difference between the current code size and the number of leftover bits, create a valid value from that combination, and save the current byte and number of unused bits.

Here is the algorithm for single code construction:

1. If necessary, read a byte from the image file. It would be faster to load some or all of the file contents and read from that stream, but this approach favors a smaller temp storage size over speed.
```
  numBytes = fread( &b1, 1, 1, inFile );
```

2. Mask off any unused bits, and assign the result as the code value.
```
  retval = ( b1 & 0x03 );
```

3. Save any remaining bits. These need to get added to the value read in the next iteration.
```
  prevValue = 0;
  prevValue = ( b1 >> 2 );
```

4. Remember how many bits were left over during this read.
```
  bitsRemaining = 6;
```

5. Track number of bytes read from file. This value may not increment for each case, depending on whether the next value can be constructed without reading the file.
```
  bytesRead++;
```

The following method fragment illustrates a portion of this algorithm. Here the logic is laid out as a set of switch constructs. The first inner case contains comments matching the algorithm just discussed. It is more verbose this way but hopefully more obvious.

```
unsigned short GifUtils::ReadCode( FILE *inFile ) {
  unsigned short retval = 0;
  unsigned short tempShort = 0;
  size_t numBytes = 0;
  Byte b1 = 0, b2 = 0;

  // Outer case depends on how many bits we need to construct a code.
  switch ( currCodeSize ) {
    case 2:
      // Inner case depends on how many bits were left unused
      // during the previous read.
      switch ( bitsRemaining ) {
        case 0:
          // Read a byte from the image file.
          numBytes = fread( &b1, 1, 1, inFile );
          retval = ( b1 & 0x03 );

          // Any remaining bits need to get added to the value read
          // in the next iteration.
          prevValue = 0;
          prevValue = ( b1 >> 2 );

          // Use this as the switch value.
          bitsRemaining = 6;

          // Are we at the end of the file?
          bytesRead++;
          break;

        case 1:
          numBytes = fread( &b1, 1, 1, inFile );
          retval = ( ( b1 & 0x01 ) << 1 ) | prevValue;
          prevValue = 0;
          prevValue = ( b1 >> 1 );
          bitsRemaining = 7;
          bytesRead++;
          break;

        case 2:
          retval = prevValue;
          prevValue = 0;
          bitsRemaining = 0;
          break;

        case 3:
          retval = ( prevValue & 0x03 );
          prevValue = ( prevValue >> 2 );
          bitsRemaining = 1;
          break;

        case 4:
          retval = ( prevValue & 0x03 );
          prevValue = ( prevValue >> 2 );
          bitsRemaining = 2;
          break;

        case 5:
          retval = ( prevValue & 0x03 );
          prevValue = ( prevValue >> 2);
          bitsRemaining = 3;
          break;

        case 6:
          retval = ( prevValue & 0x03 );
          prevValue = ( prevValue >> 2 );
          bitsRemaining = 4;
          break;

        case 7:
          retval = ( prevValue & 0x03 );
          prevValue = ( prevValue >> 2 );
          bitsRemaining = 4;
          break;

        default:
          break;
      }

      break;
```

Listing 2. Reading variable-length bit sequences.

"Outputting" a code consists of building the entire sequence represented by the code, which includes all the RGB values starting with the corresponding node, up to and including the root node. Then the entire sequence gets copied to the output image data. This is shown in listing 3. This method traces a path from a leaf node up through the parent node at the root, and generates the RGB sequence for the entire path. It checks each code against the previously-determined code for a transparent pixel, and sets a flag when appropriate.

```
void GifUtils::OutputCode( unsigned short theCode, Image &image ) {
  unsigned short prevCode = theCode;
  ImageColorEntry imageColorEntry;

  // Build the sequence as a stack of codes. Since this will result in
  // a reversed order, we then walk from the first parent down to the
  // last entry and build the final sequence.
  do {
    if ( theCode == endcode || theCode == clearcode )
      break;

    stack[ stackPointer ] = theCode;
    stackPointer++;

    prevCode = theCode;

    theCode = dictionaryTree[ theCode ].parent;
  } while ( theCode != 0 );

  // Handle off-by-one problem when walking tree to the 0-node.
  if ( prevCode > endcode ) {
    stack[ stackPointer ] = theCode;
    stackPointer++;
  }

  firstCharacter = stack[ stackPointer - 1 ];

  // Walk from the first entry to the last. Check for transparency and
  // then add each to the output stream.
  while ( stackPointer > 0 ) {
    stackPointer--;

    // If the current entry matches the transparent entry, then mark
    // this entry as transparent.
    if ( transparent && (
      dictionaryTree[ stack[ stackPointer ] ].red ==
        dictionaryTree[ transparentIndex ].red &&
      dictionaryTree[ stack[ stackPointer ] ].green ==
        dictionaryTree[ transparentIndex].green &&
      dictionaryTree[ stack[ stackPointer ] ].blue ==
        dictionaryTree[ transparentIndex ].blue ) )
      // then…
      imageColorEntry.transparent = kTransparentPixel;
    else
      imageColorEntry.transparent = kOpaquePixel;

    imageColorEntry.red =
      dictionaryTree[ stack[ stackPointer ] ].red;

    imageColorEntry.green =
      dictionaryTree[ stack[ stackPointer ] ].green;

    imageColorEntry.blue =
      dictionaryTree[ stack[ stackPointer ] ].blue;

    image.AppendToStream( imageColorEntry );
  }
}
```

Listing 3. Writing decoded color sequences.

The remaining portion of the decoding process consists of a loop that matches a byte against the next expected code. For a code that already exists in the dictionary, that code and all the colors in each parent node, up to and including the root node, get written to the (decoded) image data stream. A new code, which is one that appears in the stream before it has been defined in the dictionary, gets added to the dictionary, and then its color sequence gets written to the data stream.

```
// get first code: <code>;
code = ReadCode();
// Ignore clear codes at start.
while ( code == clearcode )
  code = ReadCode();

// Write code to dictionary.
OutputCode( code, image );

// <old> = <code>;
lastcode = code;

top:
while ( !done && bytesRead < count ) {
  // <old> <- <code>;
  lastcode = code;

  // It may be necessary to bump up the code size value.
  if ( ( nextcode >= ( unsigned int )pow( 2, currCodeSize ) ) &&
    ( currCodeSize < 12 ) ) {
      currCodeSize++;
  }

  // <code> <- next code in codestream;
  code = ReadCode();

  if ( code == endcode ) {
    done = true;
    break;
  }
  else if ( code == clearcode )
  {
    // Reinitialize dictionary with new code size.
    InitializeGifDictionary( initialCodesize );

    while ( code == clearcode )
      code = ReadCode();

    if ( code == endcode ) {
#ifdef __DEBUG__SETUP__
      printf ( " *** Found end code inside clear code loop.\n",
        code );
      printf ( " nextcode = %8d (decimal)\n", nextcode);
      printf ( " currCodeSize = %8d (decimal)\n", currCodeSize);
      printf ( " bitsRemaining = %8d (decimal)\n", bitsRemaining);
      printf ( " bytesRead = %8d (decimal)\n", bytesRead);
#endif
      done = true;
      break;
    }

    OutputCode( code, image );
  }
```

These next two blocks contain the guts of the tree-building process. Notice how an existing code gets written to the image data and then added to the dictionary, while a new code gets set as the next entry and then written to the image data.

```
  else if ( code < nextcode ) {
    OutputCode( code, image );

    dictionaryTree[ nextcode ].parent = lastcode;

    dictionaryTree[ nextcode ].red =
      dictionaryTree[ firstCharacter ].red;

    dictionaryTree[ nextcode ].green =
      dictionaryTree[ firstCharacter ].green;

    dictionaryTree[ nextcode ].blue =
      dictionaryTree[ firstCharacter ].blue;

    nextcode++;
  }
  else if ( code == nextcode )
  {
    dictionaryTree[ nextcode ].parent = lastcode;

    dictionaryTree[ nextcode ].red =
      dictionaryTree[ firstCharacter ].red;

    dictionaryTree[ nextcode ].green =
      dictionaryTree[ firstCharacter ].green;

    dictionaryTree[ nextcode ].blue =
      dictionaryTree[ firstCharacter ].blue;

    nextcode++;

    OutputCode( code, image );
  }
}

if ( ( code != endcode ) ) {
  memcpy( &count, fileByteStream + fileByteStreamOffset, 1 );

  fileByteStreamOffset += 1;

  if ( ( count != 0 ) ) {
    memcpy( &blockArray, fileByteStream + fileByteStreamOffset,
      count );

    fileByteStreamOffset += count;

    arrayByteStreamOffset = 0;
    bytesRead = 0;
    done = false;
    goto top;
  }
}
```

Listing 4. The main decoding loop.

Interlacing

Interlaced GIFs include the image data as a series of four progressive passes through the row data. This allows the image to be displayed as a “work-in-progress” while downloading or decoding: by drawing the image and using existing row data to fill in the “missing” rows, the image may be drawn in successive passes, gradually becoming clearer. This may be important when downloading over a slow connection, or when the image is large and takes a long time to decode.

The interval between rows in each of the first two passes is 8. In the first pass, rows 0, 8, 16, etc. are extracted. In pass two, rows 4, 12, 20, etc. are extracted. Pass 3 extracts rows 2, 6, 10, etc. (a 4 row interval). Pass 4 includes rows 1, 3, 5, etc. (a 2 row interval)..

Here is how to use the GIF header to ascertain whether the image is interlaced:
```
  if ( ( imageHeader.bitField & 0x40 ) > 0 )
    mInterlaced = true;
```

Listing 5 contains a method that will copy the interlaced image data to a top-to-bottom format in a new buffer. This code includes no calls to accommodate drawing while decoding.
```
void Image::Normalize( void ) {
  // Remove GIF interlacing.
  ImageColorEntry *tempBuffer =
    new ImageColorEntry[ imageStreamLength ];

  if ( tempBuffer == 0 )
    return;

  // Copy original data to temporary storage.
  // The non-interlaced data gets copied back, overwriting imageStream.
  memcpy( tempBuffer, imageStream,
    imageStreamLength * sizeof( ImageColorEntry ) );

  int interval = 0, startRow = 0, sourceRow = 0;

  for ( unsigned int pass = 1; pass < 5; pass++ ) {
    switch ( pass ) {
      case 1:
        startRow = 0;
        interval = 8;
        break;

      case 2:
        startRow = 4;
        interval = 8;
        break;

      case 3:
        startRow = 2;
        interval = 4;
        break;

      case 4:
        startRow = 1;
        interval = 2;
        break;
    }

    // The interlace constants determine the destination row.
    for ( unsigned int destRow = startRow; destRow < mBounds.height;
      destRow += interval ) {
      // Copy one row to appropriate location in image imageStream.
      // Each j value represents one pixel in the row.
      for ( unsigned int j = 0; j < mBounds.width; j++ ) {
        imageStream[ ( destRow * mBounds.width ) + j ] =
          tempBuffer[ ( sourceRow * mBounds.width ) + j ];
      }

      // The source row starts at 0 and only increases.
      sourceRow++;
    }
  }

  delete [] tempBuffer;
}
```

Listing 5. Removing interlacing.

Image display

Display is a matter of copying the pixel values to an offscreen data structure and then transferring that structure's contents onscreen. This block transfer or "blitting" is platform dependent. Listing 5 shows drawing methods for QuickDraw and GDI (Win32). These examples do not assume the interlacing has been removed from the image, and with modification may be appropriate for on-the-fly drawing. It is also simple to combine these methods and use macros to control the compilation for either platform. I separated them for this discussion.

```
void GifUtils::DrawImage( WindowPtr theWindow, GifImage *gifData ) {
  RGBColor pixelColor;
  GrafPtr oldPort;
  unsigned int i, x, y, oldX;
  unsigned int pass, interval = 0;

  GetPort( &oldPort );
  SetPort( theWindow ); /* set window to current graf port */

  x = oldX = gifData->GetBounds().x;
  y = gifData->GetBounds().y;

  if ( gifData->IsInterlaced() ) {
    i = 0;

    for ( pass = 1; pass < 5; pass++ ) {
      switch ( pass ) {
        case 1:
          y = 0;
          interval = 8;
          break;

        case 2:
          y = 4;
          interval = 8;
          break;

        case 3:
          y = 2;
          interval = 4;
          break;

        case 4:
          y = 1;
          interval = 2;
          break;
      }
      while ( y < gifData->GetBounds().height ) {
        pixelColor.red = ( gifData->GetStream()[ i ] << 8 );
        pixelColor.green = ( gifData->GetStream()[ i + 1 ] << 8 );
        pixelColor.blue = ( gifData->GetStream()[ i + 2 ] << 8 );

        SetCPixel( x, y, &pixelColor );

        i += 3;

        x++;

        if ( x >= ( oldX + gifData->GetBounds().width ) ) {
          x = oldX;

          y += interval;

        }
      }
    }
  }
  else {
    for ( i = 0; i < gifData->GetStreamLength(); i += 3 ) {
      pixelColor.red = ( gifData->GetStreamElement( i ) << 8 );
      pixelColor.green = ( gifData->GetStreamElement( i + 1 ) << 8 );
      pixelColor.blue = ( gifData->GetStreamElement( i + 2 ) << 8 );

      SetCPixel( x, y, &pixelColor );

      x++;

      if ( x >= ( oldX + gifData->GetBounds().width ) ) {
        x = oldX;

        y++;
      }
    }
  }

  SetPort( oldPort );
}
```

(a) Mac OS QuickDraw

```
void GifUtils::DrawImage( HDC hdc, GifImage *gifData ) {
  COLORREF rawColor, matchColor;
  BOOL b;
  unsigned int i, x, y, oldX;
  unsigned int pass, interval = 0;

  x = oldX = gifData->GetBounds().x;
  y = gifData->GetBounds().y;

  if ( gifData->IsInterlaced() ) {
    i = 0;

    for ( pass = 1; pass < 5; pass++ ) {
      switch ( pass ) {
        case 1:
          y = 0;
          interval = 8;
          break;

        case 2:
          y = 4;
          interval = 8;
          break;

        case 3:
          y = 2;
          interval = 4;
          break;

        case 4:
          y = 1;
          interval = 2;
          break;
      }

      while ( y < gifData->GetBounds().height ) {
        rawColor = 0x00000000;
        rawColor |= gifData->stream[ i ];
        rawColor |= ( gifData->stream[ i + 1 ] << 8 );
        rawColor |= ( gifData->stream[ i + 2 ] << 16 );
        matchColor = GetNearestColor( hdc, rawColor );
        b = SetPixelV( hdc, x, y, matchColor );

        i += 3;

        x++;

        if ( x >= ( oldX + gifData->GetBounds().width ) ) {
          x = oldX;

          y += interval;


        }
      }
    }
  }
  else {
    for ( i = 0; i < gifData->GetStreamLength(); i += 3 ) {
      rawColor = 0x00000000;
      rawColor |= gifData->GetStreamElement( i );
      rawColor |= ( gifData->GetStreamElement( i + 1 ) << 8 );
      rawColor |= ( gifData->GetStreamElement( i + 2 ) << 16 );
      matchColor = GetNearestColor( hdc, rawColor );
      b = SetPixelV( hdc, x, y, matchColor );

      x++;

      if ( x >= ( oldX + gifData->GetBounds().width ) ) {
        x = oldX;

        y++;
      }
    }
  }
}
```

(b) Win32 GDI

Listing 5. Drawing on several platforms.

The patent

GIF, though popular for many years, lies at the center of a controversy that illustrates the clash between free and controlled software. Terry Welsh, who worked at Sperry (now Unisys), applied for a patent on the LZW process in the mid-1980s. CompuServe adopted GIF as its standard for online graphics files without realizing the compression algorithm was covered under a pending patent. Years passed before Unisys began to assert its rights to license fees. And when it did, confusion reigned. What was the fee structure? Should both GIF readers and writers require LZW licenses? What about big software houses vs. freeware authors? Unisys' responses were sometimes inconsistent and it appears (judging from the various online articles cited below) that the company did not understand the dynamics of the software industry.

This was before the Internet as we know it today: in the early 1990s the online services dominated the growing home market, and CompuServe was at its peak. GIF provided a convenient format for downloading compressed images over dialup connections, because most images were not of photo quality and could reasonably be expected to contain less than 256 colors.

The actions of Unisys in attempting to enforce the LZW patent by levying license fees against developers big and small led to the adoption of alternate image formats, PNG in particular. Yet GIF had already taken hold and remains a popular format.

What makes this more relevant is that the LZW patent expires this year during MacHack. The GIF patent expires in the United States at midnight on June 19, 2003. Overseas it expires in June 2004. The code contained in this paper may actually be legal by the time you read it.

If you are interested in the history of the patent issues or individual developers' headaches in trying to obtain legitimate licenses then consider reading some of the following articles (obtained via a Google search):

http://patft.uspto.gov The U.S. Patent Office site. The LZW patent number is 4,558,302.

http://www.unisys.com/about_unisys/lzw/ LZW Patent and Software Information.

http://cloanto.com/users/mcb/19950127giflzw.html **The GIF Controversy: A Software Developer's Perspective.**

http://www.serverobjects.com/lzw.html **AspImage, LZW and the Unisys Patent on GIF/LZW. Another developer's attempt to do the right thing.**

http://news.com.com/2100-1023-239414.html?legacy=cnet&tag=st.ne.1002.tgif.1005-200-1713278?st.ne.fd.gif.f **c/net article that includes statements from Unisys.**

http://burnallgifs.org/ **A project of the League for Programming Freedom.**

Conclusion

Adding custom support for GIFs allows you finer control over the decoding and display process. The decoding algorithm can be implemented in less than 20k of object code (before optimizing). In spite of other image file formats GIF remains popular and useful for certain applications.

Bibliography

[Kientzle] Kientzle, Tim. Internet File Formats. The Coriolis Group, Inc. 1995.

[Miano] Miano, John. Compressed Image File Formats. ACM Press. 1999.