OTH Amberg-Weiden
East Bavarian Technical University Amberg-Weiden
Germany
Faculty of Electronic-Engineering, Media and Computer Science
Dieter Meiller | Computer Science and Media
d.meiller@oth-aw.de

IATUL 2015

Ostbayerische Technische Hochschule
Amberg-Weiden

Von Neumann Data

Abstract

In this work we present a new approach for storing digital objects: We suggest saving program code for viewing the file together with payload inside the same file. The whole code should be human readable. Then, the file format will be comprehensible in future. In addition, the file has the capability to display itself.

Problem and solution

One of the main problems in the field of long-term preservation is the following: It is not sure that digitally stored documents can be reproduced as *Dissemination Information Packages* (DIP) [1, P. 4-36] in future times. Apart from hardware problems, it is important to have a working program available that is capable to interpret and reproduce the data contained in the document correctly.
There are several methods to deal with the problem:

- Software environments including the application programs will be archived, too.
- Documents will be converted to an up-to-date file format.
- File specifications will be archived, too, so another program to reproduce the file can be written in future.

Main problem is that the file and the way to read it are stored separately.
We suggest to store the program for reproducing the data (*Representation Rendering Software* [1, P. 4-24]) in combination with the data inside the same file. We assume that it will be possible to read character sets like ANSI and that English will be known when the file will be reproduced in future. All data, the actual file content and the code should be available in textual, human reada-ble form. The code should be executable to view or replay the file. Moreover, it should be an explanation how to do this (*Access software* [1, P. 4-25]). Therefore, the code must be a turing-complete subset of a programming language which is similar readable as pseudo code. Furthermore, some basic commands for visual output should be postulated. Examples: *point(x,y)* or *text(t,x,y)* to draw a point or a text t on a surface at (x,y). Increasing file size should not be a reason against this approach: We claim that in future file sizes will increase stronger than sizes of codes to interpret them. The code should be generated as well, so that it is not necessary to store all the logic of the producing source program inside the file. Like in a von Neumann computer, it could be an interesting projection to store the logic together with the data inside the files. Then, they could be able to reproduce themselves.

Sample implementation

To prove the concept, we made a sample implementation. The aim was to clarify the concept with a descriptive example. So, we built a GIF image that is able to display itself. We assume that it is possible to generate the following code automatically. It should be possible to convert any GIF image into similar code by using special converter programs. The code of the example was written in a subset of Python, a language with a clear syntax. We didn´t use any object-oriented features and no function definitions. Only imperative programming was used to write code for displaying GIF images.

In line 1 all bytes of the image formatted according to GIF specification are listed. In our example, bytes are written as hexadecimal numbers. In future implementations, data could be encoded in a more efficient, but human-readable way, with the base64 method [2]. In lines 2-8 important variables, representing significant parts of the data, are defined. Line 4 specifies width and height of the surface to draw on.
In this section of code (Lines 9-39), LZW decompression is implemented. It is the largest part of the program code. However, it has a length of only 30 lines of code. The bits extracted from stored data (lines 9-12) will be decompressed in a loop (lines 24-38). It is difficult to write code for the LZW algorithm according to formal specifications [3]. Our code is the implementation of the algorithm described by a web tutoial [4]. When no such instructions will be available in future, it could be difficult to interpret data. It will be helpful to get the implementation together with the data. Even when the program is not executable, it gives an example how to do it, especially when the code is self-explaining [5].

The third part of code draws single pixels onto a surface. Each of the extracted values represents an entry in the color table. The appropriate color for drawing will be specified in line 47, and pixels will be drawn with the command in line 48. The whole code contains several atomic commands for handling data structures and for drawing. Examples: point(xpos,ypos) draws a point on the surface at (xpos,ypos); range(3) generates an array [0, 1, 2]. These commands could be explained in a fourth section: documentation. Drawing commands are similar to those from the processing language [6], Commands for operating with data are similar to those from python. A parser could generate the needed documentation after writing the program code. This code is executable and generates the expected image. The code is human readable. Probably, it will be in future, too.

```
1: data = [   0x47, 0x49, 0x46, 0x38, 0x39, 0x61, 0x0A,
              0x00, 0x0A, 0x00, 0x91, 0x00, 0x00, 0xFF,
              0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
              0xFF, 0x00, 0x00, 0x00, 0x21, 0xF9, 0x04,
              0x00, 0x00, 0x00, 0x00, 0x00, 0x2C, 0x00,
              0x00, 0x00, 0x00, 0x0A, 0x00, 0x0A, 0x00,
              0x00, 0x02, 0x16, 0x8C, 0x2D, 0x99, 0x87,
              0x2A, 0x1C, 0xDC, 0x33, 0xA0, 0x02, 0x75,
              0xEC, 0x95, 0xFA, 0xA8, 0xDE, 0x60, 0x8C,
              0x04, 0x91, 0x4C, 0x01, 0x00, 0x3B]
2: img_width = lsd[1]*256+lsd[0]
3: img_height = lsd[3]*256+lsd[2]
4: size(img_width,img_height)
5: color_table = data[13:24]
6: dict_byte_size = data[43]
7: data_size = data[44]
8: bytes = data[45:45+data_size]
```

Data

```
 9: lzw = []
10: for b in bytes:
11:     for i in range(8):
12:         lzw = lzw + [((b >> i) & 1)]
13: code_byte_size = dict_byte_size+1
14: dict_size = 2 ** (dict_byte_size) + 2
15: eof = dict_size -1
16: dictionary = dict((i, [i]) for i in
range(dict_size))
17: uncompressed = []
18: for i in range(code_byte_size):
19:     lzw.pop(0)
20: code=0
21: for i in range(code_byte_size):
22:     code = code + lzw.pop(0) * 2**i
23: uncompressed = uncompressed + dictionary[code]
24: while len(lzw) >= code_byte_size:
25:     code_1 = code
26:     code = 0
27:     for i in range(code_byte_size):
28:         code = code + lzw.pop(0) * 2**i
20:     if code in dictionary:
30:         uncompressed = uncompressed +
                dictionary[code]
31:         k = dictionary[code][0]
32:     else:
33:         k = dictionary[code_1][0]
34:         uncompressed = uncompressed +
                (dictionary[code_1] + [k])
35:     if len(dictionary) == 2**(code_byte_size)-1 :
36:         code_byte_size = code_byte_size + 1
37:     dictionary[dict_size] = dictionary[code_1] + [k]
38:     dict_size += 1
```

Decompression code

```
39: pos = 0
40: for n in uncompressed:
41:     if n == eof: break
42:     xpos = pos % width
43:     ypos = pos / width
44:     r = color_table[n*3]
45:     g = color_table[n*3+1]
46:     b = color_table[n*3+2]
47:     stroke(r,g,b)
48:     point(xpos,ypos)
49:     pos = pos + 1
```

Display code

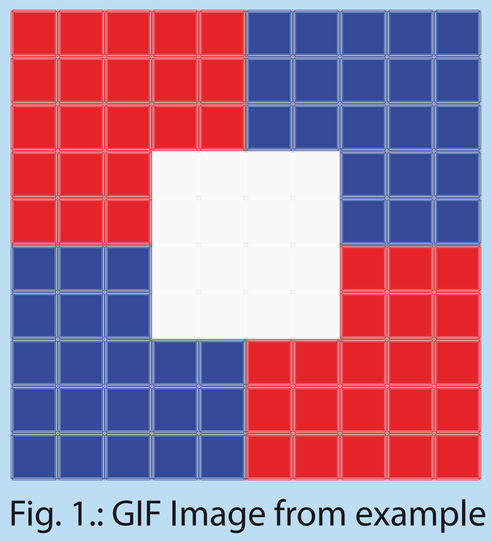

Fig. 1.: GIF Image from example

References

1. Consultative Committee for Space Data Systems.
   Reference Model for an Open Archival Information System (OAIS). Washington, DC: CCSDS Secretariat, p. 1-1. (2012)
2. IETF: The Base16, Base32 and Base64 Data Encodings.
   https://tools.ietf.org/html/rfc4648 (2006)
3. W3C.: Graphics Interchange Format (sm) Version 89a. http://www.w3.org/Graphics/GIF/spec-gif89a.txt (1987)
4. Flickinger, M.: Project: What´s in a GIF. http://www.matthewflickinger.com/lab/whatsinagif/index.html (1/24/2005)
5. Knuth, D. E.: Literate Programming. In: The Computer Journal. 27, Nr. 2, ,97–111 (1984)
6. Fry, B.: Visualizing Data. O'Reilly. (2007)