GIF Steganography from First Principles

dtm.uk/gif-steganography

DTM September 10, 2023

DTM

Sep 10, 2023 • 26 min read

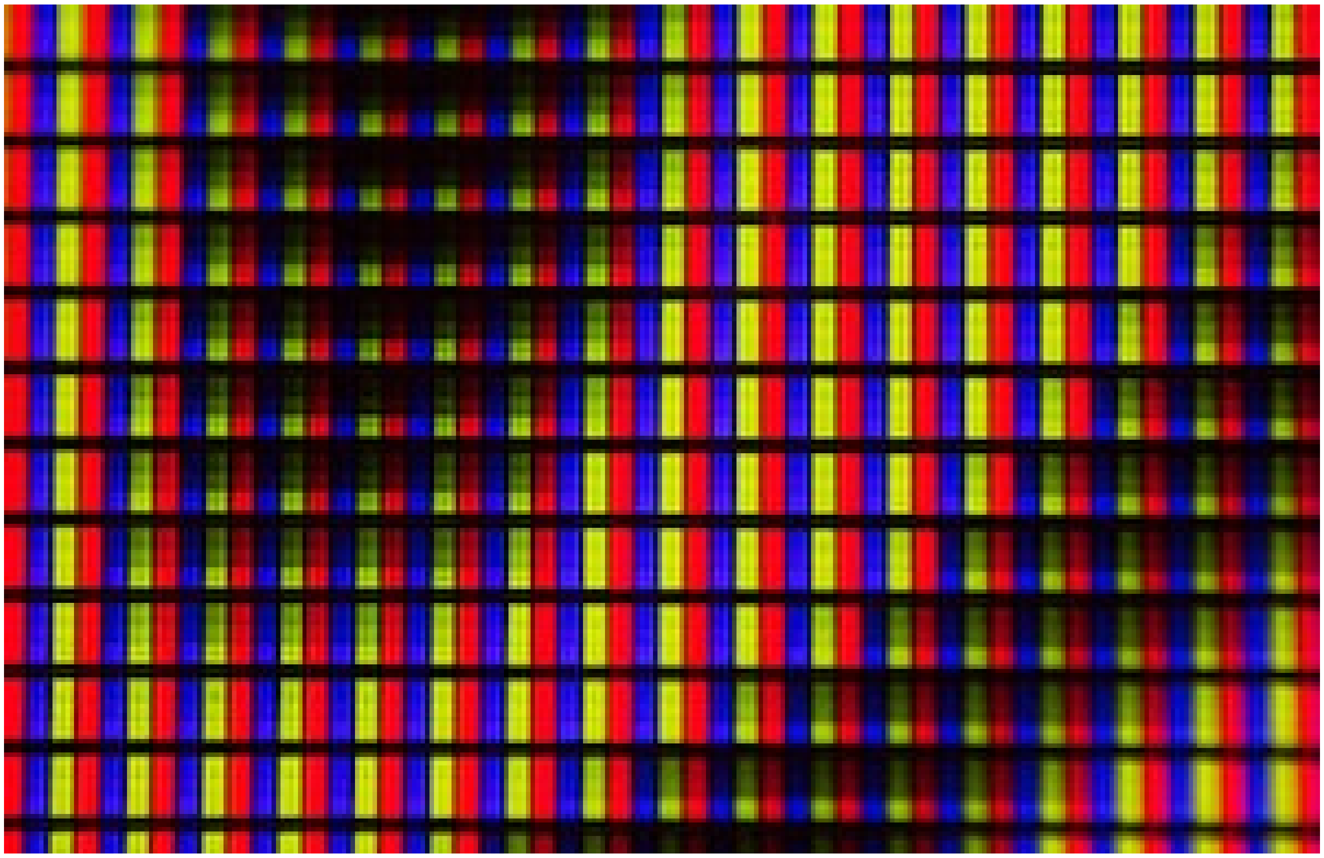


Photo by Michael Maasen / Unsplash

Inspiration

Adam's blog on PNG steganography inspired me to start a project in a similar vein but focused on GIF files instead. Although I have previously written several .NET-based steganography tools previously, I have typically relied on existing libraries for the steganography part without delving into the fundamentals. It was great to step through the file format with scrappy code examples before refactoring it into a more complete solution, making it easier to understand and add the steganography in. Once you understand the file format it becomes clear just how many opportunities to hide data are there.

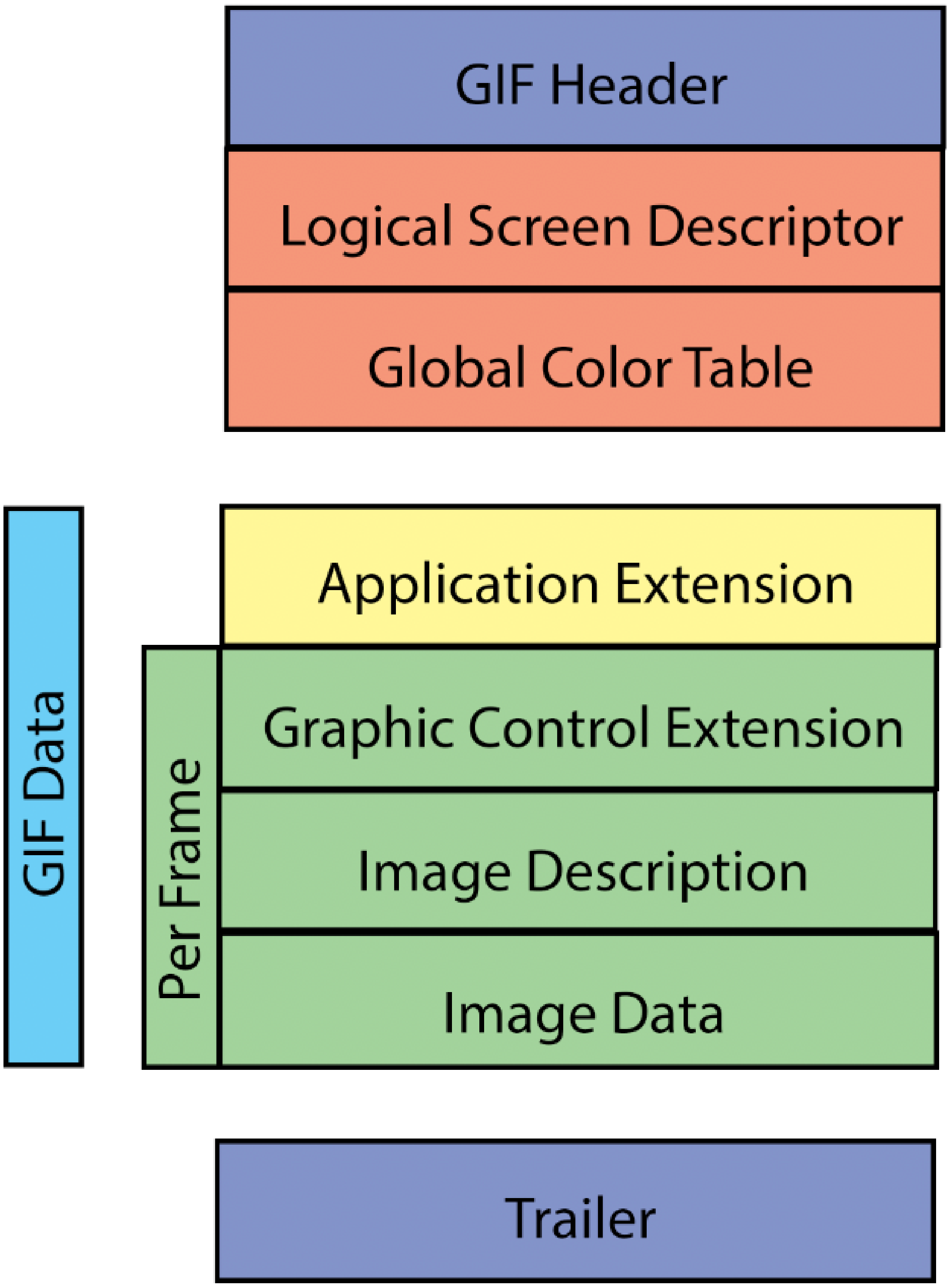
Why a GIF file?

I was intrigued by the possibility of hiding information across multiple frames. Additionally, animated GIF files are generally large files. By the end of this blog, you will be able to hide data across multiple frames of an animation which means that you can store more data with less visual artefacts.

How is a GIF Structured?

Before diving into the steganography part, it's crucial to understand the anatomy of a GIF file. A GIF (Graphics Interchange Format) is a bitmap image format that supports animations.

After looking at a bunch of different GIFs I made this simplified diagram showing a rough structure for an animated GIF.



You can read more in the latest spec here. We will step the most important parts below.

GIF Header

The GIF header tells you which version of the GIF standard the file uses. The two main versions are GIF87a and GIF89a.

Here is an example header:

```
00000000: 4749 4638 3961 2c01 2c01 f700 00af 6b37  GIF89a,.,.....k7
```

Logical Screen Descriptor

Think of the logical screen descriptor as the stage where your GIF will be displayed. It sets the size of the "stage" by specifying the screen width and height. It also tells you the background color of the stage.

```
struct LogicalScreenDescriptor {
uint16_t width;             // Logical Screen Width (2 bytes)
uint16_t height;            // Logical Screen Height (2 bytes)
uint8_t packed;             // Packed Fields (1 byte)
uint8_t backgroundColor;    // Background Color Index (1 byte)
uint8_t pixelAspectRatio;   // Pixel Aspect Ratio (1 byte)
};
```

Here’s is some Python code to read the logical screen descriptor for a GIF file:

```
def read_logical_screen_descriptor(file_path):
    with open(file_path, 'rb') as f:
        # Skip the GIF header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor
        screen_width = int.from_bytes(f.read(2), 'little')
        screen_height = int.from_bytes(f.read(2), 'little')

        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0b10000000) >> 7
        color_res = (packed_fields & 0b01110000) >> 4
        sort_flag = (packed_fields & 0b00001000) >> 3
        gct_size = packed_fields & 0b00000111

        bg_color_index = int.from_bytes(f.read(1), 'little')
        pixel_aspect_ratio = int.from_bytes(f.read(1), 'little')

        print(f"Logical Screen Width: {screen_width}")
        print(f"Logical Screen Height: {screen_height}")
        print(f"Global Color Table Flag: {gct_flag}")
        print(f"Color Resolution: {color_res}")
        print(f"Sort Flag: {sort_flag}")
        print(f"Size of Global Color Table: {gct_size}")
        print(f"Background Color Index: {bg_color_index}")
        print(f"Pixel Aspect Ratio: {pixel_aspect_ratio}")
```

Ok let’s grab a GIF and give this a try:

```
wget https://media.giphy.com/media/s51QoNAmM6dkWcSC0P/giphy.gif
```

Running the above function:

```
>>> read_logical_screen_descriptor('giphy.gif')

Logical Screen Width: 480
Logical Screen Height: 450
Global Color Table Flag: 1
Color Resolution: 7
Sort Flag: 0
Size of Global Color Table: 7
Background Color Index: 255
Pixel Aspect Ratio: 0
```

Global Color Table

The Global Color Table is a table of RGB values that represent the overall color palette for the GIF. Below is the 256 colors extracted from the downloaded GIF - this was rendered in HTML generated with the below code which parses out the global color table:

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 2
9 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 8
0 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 1
04 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 1
59 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 17
7 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213
214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 2
32 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 25
0 251 252 253 254 255**

```
import struct

def read_global_color_table(file_path):
    with open(file_path, 'rb') as f:
        # Skip GIF Header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor
        screen_width, screen_height = struct.unpack("<HH", f.read(4))
        packed_fields = struct.unpack("<B", f.read(1))[0]
        bg_color_index = struct.unpack("<B", f.read(1))[0]
        pixel_aspect_ratio = struct.unpack("<B", f.read(1))[0]

        # Check if Global Color Table exists (bit 7 of packed_fields)
        gct_flag = (packed_fields & 0b10000000) >> 7

        if gct_flag:
            # Determine the size of the Global Color Table
            gct_size_bits = packed_fields & 0b00000111
            gct_size = 2 ** (gct_size_bits + 1)

            print(f"Global Color Table Size: {gct_size} colors")

            # Read Global Color Table
            gct_data = f.read(3 * gct_size)  # Each color is 3 bytes (RGB)

            # Create a list of RGB triplets
            gct_colors = [(gct_data[i], gct_data[i + 1], gct_data[i + 2]) for i in
range(0, len(gct_data), 3)]

            # Print or use Global Color Table as needed
            for i, color in enumerate(gct_colors):
                print(f"Color {i}: R={color[0]} G={color[1]} B={color[2]}")
            return gct_colors

        else:
            print("No Global Color Table present.")
            return []

def rgb_to_hex(r, g, b):
    return "#{:02x}{:02x}{:02x}".format(r, g, b)

colors = read_global_color_table('giphy.gif')

with open("colours.html", "w") as fh:
    i = 0
    fh.write("<div style=\"width: 512px; word-wrap: break-word\">")
    for color in colors:
        html_code = rgb_to_hex(color[0], color[1], color[2])
        fh.write(f"<font color={html_code}><b>{i}</b></font>&nbsp;")
        i += 1
    fh.write("</div>")
```

Running this for our GIF gives:

```
Global Color Table Size: 256 colors
Color 0: R=17 G=8 B=7
Color 1: R=19 G=15 B=21
Color 2: R=25 G=10 B=7
Color 3: R=25 G=16 B=22
Color 4: R=26 G=20 B=20
Color 5: R=26 G=20 B=26
Color 6: R=29 G=23 B=30
Color 7: R=30 G=17 B=15
Color 8: R=30 G=20 B=25
Color 9: R=34 G=27 B=32
Color 10: R=36 G=25 B=24
--snip--
```

GIF Data

The main data section of a GIF typically starts with an Application Extension block:

Application Extension

```
struct GIFApplicationExtension {
    uint8_t extensionIntroducer;  // Should be 0x21
    uint8_t applicationLabel;     // Should be 0xFF
    uint8_t blockSize;            // Typically 11
    uint8_t appData[APP_EXT_BLOCK_SIZE];  // Application Identifier and
Authentication Code
};

struct GIFDataSubBlock {
    uint8_t size;  // 0 <= size <= 255
    uint8_t* data; // Dynamic array to hold data
};
```

Here’s a Python function scan through and find and read the Application Extension:

```
def read_application_extension(file_path):
    with open(file_path, 'rb') as f:
        # Skip the GIF header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor (7 bytes)
        f.read(4)  # Skip screen width and height
        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0x80) >> 7
        gct_size = packed_fields & 0x07
        f.read(2)  # Skip background color index and pixel aspect ratio

        # Skip Global Color Table if it exists
        if gct_flag:
            gct_length = 3 * (2 ** (gct_size + 1))
            f.read(gct_length)

        # Loop through the blocks to find the first Application Extension
        while True:
            block_type = f.read(1)
            if not block_type:
                raise EOFError("Reached end of file without finding an Application
Extension.")

            if block_type == b'\x21':  # Extension Introducer
                next_byte = f.read(1)
                if next_byte == b'\xFF':  # Application Extension Label
                    # Process Application Extension
                    block_size = int.from_bytes(f.read(1), 'little')
                    if block_size == 11:  # Typically 11 for Application Extension
                        app_identifier = f.read(8).decode('ascii')
                        app_auth_code = f.read(3).hex()

                        print(f"Block Size: {block_size}")
                        print(f"Application Identifier: {app_identifier}")
                        print(f"Application Authentication Code: {app_auth_code}")

                        # Reading and displaying sub-blocks
                        while True:
                            sub_block_size = int.from_bytes(f.read(1), 'little')
                            if sub_block_size == 0:
                                break
                            else:
                                sub_block_data = f.read(sub_block_size)
                                print(f"Sub-block Size: {sub_block_size}")
                                print(f"Sub-block Data: {sub_block_data.hex()}")

                        break

                else:  # Skip other types of extensions
                    extension_length = int.from_bytes(f.read(1), 'little')
                    f.read(extension_length)
```

For our GIF from Giphy we get:

```
Block Size: 11
Application Identifier: NETSCAPE
Application Authentication Code: 322e30
Sub-block Size: 3
Sub-block Data: 010000
```

The last two bytes indicate how many times the GIF file should loop up to 65,535 times. 0 indicates unlimited looping.

Per Frame

Image Descriptor

This is like a snapshot of each scene or frame in your GIF. It tells you where the top-left corner of the image starts and how big the image is.

```
struct GIFImageDescriptor {
    uint16_t left_position;  // X position of image in the logical screen
    uint16_t top_position;   // Y position of image in the logical screen
    uint16_t width;          // Width of the image
    uint16_t height;         // Height of the image
    uint8_t  packed_field;   // Packed fields that indicate various settings, such as
local color table presence
                             // bit 0:    Local Color Table Flag (1 = Local Color
Table Present)
                             // bit 1:    Interlace Flag
                             // bit 2:    Sort Flag
                             // bit 3-4:  Reserved
                             // bit 5-7:  Size of Local Color Table (if present)
}
```

Let’s loop until we find the first Image Descriptor:

```
def read_gif_image_descriptor(file_path):
    with open(file_path, 'rb') as f:
        # Skip the GIF header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor (7 bytes)
        f.read(4)  # Skip screen width and height
        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0x80) >> 7
        gct_size = packed_fields & 0x07
        f.read(2)  # Skip background color index and pixel aspect ratio

        # Skip Global Color Table if it exists
        if gct_flag:
            gct_length = 3 * (2 ** (gct_size + 1))  # 3 bytes for each color
            f.read(gct_length)

        # Loop through the blocks to find the first Image Descriptor
        while True:
            block_type = f.read(1)
            if not block_type:
                raise EOFError("Reached end of file without finding an image
descriptor.")

            if block_type == b'\x2C':
                # Read and process Image Descriptor (9 bytes)
                left_position = int.from_bytes(f.read(2), 'little')
                top_position = int.from_bytes(f.read(2), 'little')
                width = int.from_bytes(f.read(2), 'little')
                height = int.from_bytes(f.read(2), 'little')
                packed_field = int.from_bytes(f.read(1), 'little')

                local_color_table_flag = (packed_field & 0x80) >> 7
                interlace_flag = (packed_field & 0x40) >> 6
                sort_flag = (packed_field & 0x20) >> 5
                reserved = (packed_field & 0x18) >> 3
                local_color_table_size = packed_field & 0x07

                print(f"Left Position: {left_position}")
                print(f"Top Position: {top_position}")
                print(f"Image Width: {width}")
                print(f"Image Height: {height}")
                print(f"Local Color Table Flag: {local_color_table_flag}")
                print(f"Interlace Flag: {interlace_flag}")
                print(f"Sort Flag: {sort_flag}")
                print(f"Reserved: {reserved}")
                print(f"Size of Local Color Table: {local_color_table_size}")

                break
```

Running this on our GIF we get for the first frame:

```
Left Position: 0
Top Position: 0
Image Width: 480
Image Height: 450
Local Color Table Flag: 0
Interlace Flag: 0
Sort Flag: 0
Reserved: 0
Size of Local Color Table: 0
```

Local Color Table

The Local Color Table (LCT) allows each frame to have its own color palette. Otherwise each frame will use the Global Color Table. We will skip past this for now as it looks the same as the GCT and will be parsed if the LCT flag is set in the Image Descriptor.

Graphics Control Extension

This is the director of your GIF movie. It tells each frame how long to stay on screen.

```
struct GraphicsControlExtension {
    uint8_t block_size;           // Block Size: Fixed as 4 bytes (should be 0x04)
    uint8_t packed;               // Packed Field: Various flags
    uint16_t delay_time;          // Delay Time: Time for this frame in hundredths of
a second
    uint8_t transparent_color;    // Transparent Color Index
    uint8_t block_terminator;     // Block Terminator: Fixed as 0 (0x00)
};
```

Now let’s scan for the graphics control extension in Python:

```
def read_graphics_control_extension(file_path):
    with open(file_path, 'rb') as f:
        # Skip the GIF header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor (7 bytes)
        f.read(4)  # Skip screen width and height
        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0x80) >> 7
        gct_size = packed_fields & 0x07
        f.read(2)  # Skip background color index and pixel aspect ratio

        # Skip Global Color Table if it exists
        if gct_flag:
            gct_length = 3 * (2 ** (gct_size + 1))
            f.read(gct_length)

        # Loop through the blocks to find the first Graphics Control Extension
        while True:
            block_type = f.read(1)
            if not block_type:
                raise EOFError("Reached end of file without finding a Graphics
Control Extension.")

            if block_type == b'\x21':  # Extension Introducer
                next_byte = f.read(1)
                if next_byte == b'\xF9':  # Graphic Control Label
                    # Process Graphics Control Extension
                    block_size = int.from_bytes(f.read(1), 'little')
                    packed_fields = int.from_bytes(f.read(1), 'little')
                    disposal_method = (packed_fields & 0b00011100) >> 2
                    user_input_flag = (packed_fields & 0b00000010) >> 1
                    transparent_color_flag = packed_fields & 0b00000001
                    delay_time = int.from_bytes(f.read(2), 'little')
                    transparent_color_index = int.from_bytes(f.read(1), 'little')
                    block_terminator = int.from_bytes(f.read(1), 'little')

                    print(f"Block Size: {block_size}")
                    print(f"Disposal Method: {disposal_method}")
                    print(f"User Input Flag: {user_input_flag}")
                    print(f"Transparent Color Flag: {transparent_color_flag}")
                    print(f"Delay Time: {delay_time}")
                    print(f"Transparent Color Index: {transparent_color_index}")
                    print(f"Block Terminator: {block_terminator}")

                    break

                else:  # Skip other types of extensions
                    extension_length = int.from_bytes(f.read(1), 'little')
                    f.read(extension_length)
```

Running this on our GIF we get:

```
Block Size: 4
Disposal Method: 1
User Input Flag: 0
Transparent Color Flag: 1
Delay Time: 7
Transparent Color Index: 255
Block Terminator: 0
```

Image Data

This containts the actual picture data for each frame, usually compressed to save space.

```
struct ImageData{
    uint8_t LZWMinimumCodeSize; // The minimum LZW code size
};

struct SubBlock{
    uint8_t size;  // 0 <= size <= 255
    uint8_t* data; // Dynamic array to hold data
} ;
```

To scan for and dump image data let’s use this Python function:

```
def read_image_data(file_path):
    with open(file_path, 'rb') as f:
        # Skip the GIF header (6 bytes: GIF87a or GIF89a)
        f.read(6)

        # Read Logical Screen Descriptor (7 bytes)
        f.read(4)  # Skip screen width and height
        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0x80) >> 7
        gct_size = packed_fields & 0x07
        f.read(2)  # Skip background color index and pixel aspect ratio

        # Skip Global Color Table if it exists
        if gct_flag:
            gct_length = 3 * (2 ** (gct_size + 1))
            f.read(gct_length)

        # Loop through the blocks to find the first Image Descriptor
        while True:
            block_type = f.read(1)
            if not block_type:
                raise EOFError("Reached end of file without finding an Image
Descriptor.")

            if block_type == b'\x2C':  # Image Descriptor
                # Skip the Image Descriptor and focus on Image Data
                f.read(9)  # Skip the next 9 bytes

                # Read the LZW Minimum Code Size
                LZW_min_code_size = int.from_bytes(f.read(1), 'little')
                print(f"LZW Minimum Code Size: {LZW_min_code_size}")

                # Reading and displaying sub-blocks
                while True:
                    sub_block_size = int.from_bytes(f.read(1), 'little')
                    if sub_block_size == 0:
                        break
                    else:
                        sub_block_data = f.read(sub_block_size)
                        print(f"Sub-block Size: {sub_block_size}")
                        print(f"Sub-block Data: {sub_block_data.hex()}")

                break

            elif block_type == b'\x21':  # Extension Introducer
                # Skip other types of extensions
                f.read(1)  # Read the label
                extension_length = int.from_bytes(f.read(1), 'little')
                f.read(extension_length)
                while True:
                    sub_block_size = int.from_bytes(f.read(1), 'little')
                    if sub_block_size == 0:
```

```
                break
            else:
                f.read(sub_block_size)
```

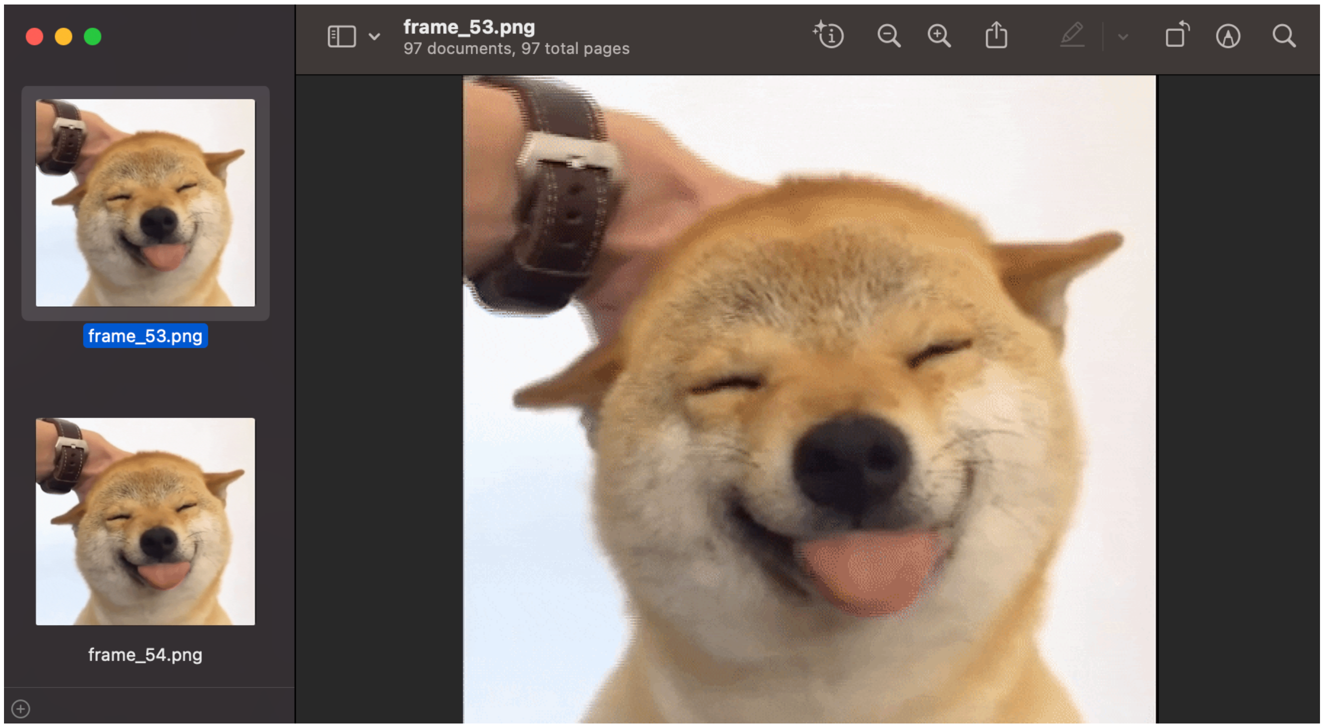
We get the following (truncated for brevity):

```
LZW Minimum Code Size: 8
Sub-block Size: 255
Sub-block Data:
0021b0d9c46a132f579776a4d89022058a872d5a9c904163860a1327326ad478b1a38a152041ce1839b28
6491d4692e8a8926a1dbf973063ca9c49b3a6cd9b3863f6a3b793674d9f3d830a054a3427cda247911a5d
9a54a9d3a750a3ea1bda542ad5ab56874edd2aafabd7aff6c28a1d4bb6ac59b068cdee5bcbb6edbab770e
3ba9d4bf79dddbb78dde985b7f7aebc04ed041274b509e19286881f9a904871468c8d90217f9c1c72068b
91974ba6ac32a825d3cfa03f631dad9574d6a2a64fab5e4d34356bd45c63bb362dbbb66dd8b6d3aaddcd9
b6e6fb2f5820bcf3bbc38f1e3c8fb2a5fee17b0e04ebc0a5f7ae322f1431412279abc88b17b648e94c387
Sub-block Size: 255
Sub-block Data:
34d9a44a931a4d068d8b1abafdcb9df77ace9c2d541fb9dbf45febdfcfff3557dca5e517207e030ea8db8
16af9b38f820cfaf6db837519275772143267215e6139371074d2258203620ea140824430a8301277dda5
e85d461e7d74c245218994847926cd618c4bfe40e59e68f16d958f3e3f0a58a09044f667e4910016595b9
24a32991b82504228e56f125668e59558daf55760686cd20961ae2cd2c8611634741d0a1849f4d1082ab6
6982089179f4e29c275ca6430d2c88500532f6f013568eeced58557c3df6a8536b4d268aa8a248367a248
18b1209e9935156ea606f0d4e59e5a69c667921725b3e77902b08bda11043253834029a18a1c8e6abafb6
Sub-block Size: 255
Sub-block Data:
f95d9c75d6908449490ca20da0bc062ae87c3d197ae87b8c166baca3c8f63769a4f42d4ba9a59a460b218
51376eae9a77c6919aa199b74cbcb22618e99429929942022092330e66aac1f8cd0aebb70aaf85dbc27e8
30c77922d430883a7ff6aae3aff4e443aca1c2126bf0b14e229cecc28d3a8bf0b395422bedc4d6567bedc
51886da2587a48aa94675a89a8b2e9b159d300208b0c2fa2ebbf1c2e972cb75563147127526a1c936dc68
a3cecee38cb30ea000e7049f8f441fac70c24733acf4d20f1b18b1c453661a6dc554639ced711906c6ad4
1602264c6a9a88a28620b37a87402ca29bb9b769bf1b6c982bd55d480420c4f58620a33d248934ade3b5f
..........
```

Let’s validate we are on the right track here and dump some frames.

```
from PIL import Image
import numpy as np
from collections import deque

def lzw_decode(min_code_size, compressed):
    clear_code = 1 << min_code_size
    eoi_code = clear_code + 1
    next_code = eoi_code + 1
    current_code_size = min_code_size + 1

    dictionary = {i: [i] for i in range(clear_code)}
    dictionary[clear_code] = []
    dictionary[eoi_code] = None

    def read_bits(bit_count, bit_pos, data):
        value = 0
        for i in range(bit_count):
            byte_pos = (bit_pos + i) // 8
            bit_offset = (bit_pos + i) % 8
            if data[byte_pos] & (1 << bit_offset):
                value |= 1 << i
        return value

    bit_pos = 0
    data_length = len(compressed) * 8
    output = deque()
    current_code = None

    while bit_pos + current_code_size <= data_length:
        code = read_bits(current_code_size, bit_pos, compressed)
        bit_pos += current_code_size

        if code == clear_code:
            current_code_size = min_code_size + 1
            next_code = eoi_code + 1
            dictionary = {i: [i] for i in range(clear_code)}
            dictionary[clear_code] = []
            dictionary[eoi_code] = None
            current_code = None
        elif code == eoi_code:
            break
        else:
            if code in dictionary:
                entry = dictionary[code]
            elif code == next_code:
                entry = dictionary[current_code] + [dictionary[current_code][0]]
            else:
                raise ValueError(f"Invalid code: {code}")

            output.extend(entry)

            if current_code is not None:
                dictionary[next_code] = dictionary[current_code] + [entry[0]]
                next_code += 1

                if next_code >= (1 << current_code_size):
                    if current_code_size < 12:
                        current_code_size += 1

            current_code = code

    return list(output)

def read_and_dump_frames(file_path):
    frame_counter = 0
    global_frame_data = None  # To hold the entire frame canvas

    with open(file_path, 'rb') as f:
        f.read(6)  # Skip the GIF header

        global_width = int.from_bytes(f.read(2), 'little')
        global_height = int.from_bytes(f.read(2), 'little')

        packed_fields = int.from_bytes(f.read(1), 'little')
        gct_flag = (packed_fields & 0x80) >> 7
        gct_size = packed_fields & 0x07
        f.read(2)  # Skip remaining fields

        if gct_flag:
            gct_length = 3 * (2 ** (gct_size + 1))
            global_color_table = np.array(list(f.read(gct_length))).reshape(-1, 3)

        # Initialize global_frame_data to zeros
        global_frame_data = np.zeros((global_height, global_width, 3),
dtype=np.uint8)

        while True:
            block_type = f.read(1)
            if not block_type:
                print("Reached end of file.")
                break

            if block_type == b'\x2C':  # Image Descriptor
                left_position = int.from_bytes(f.read(2), 'little')
                top_position = int.from_bytes(f.read(2), 'little')
                width = int.from_bytes(f.read(2), 'little')
                height = int.from_bytes(f.read(2), 'little')
                packed_field = int.from_bytes(f.read(1), 'little')

                interlace_flag = (packed_field & 0x40) >> 6
                local_color_table_flag = (packed_field & 0x80) >> 7
                if local_color_table_flag:
                    lct_size = packed_field & 0x07
                    lct_length = 3 * (2 ** (lct_size + 1))
                    local_color_table = 
np.array(list(f.read(lct_length))).reshape(-1, 3)
                else:
                    local_color_table = global_color_table

                LZW_min_code_size = int.from_bytes(f.read(1), 'little')
                compressed_data = bytearray()
                while True:
                    sub_block_size = int.from_bytes(f.read(1), 'little')
                    if sub_block_size == 0:
                        break
                    compressed_data += f.read(sub_block_size)

                decoded_data = lzw_decode(LZW_min_code_size, compressed_data)
                frame_data = np.zeros((height, width, 3), dtype=np.uint8)

                if interlace_flag:
                    interlace_order = []
                    interlace_order.extend(range(0, height, 8))
                    interlace_order.extend(range(4, height, 8))
                    interlace_order.extend(range(2, height, 4))
                    interlace_order.extend(range(1, height, 2))

                    reordered_data = [None] * height

                    for i, row in enumerate(interlace_order):
                        start_index = row * width
                        end_index = start_index + width
                        reordered_data[row] = decoded_data[start_index:end_index]

                    decoded_data = [pixel for row_data in reordered_data if row_data 
is not None for pixel in row_data]

                for i, pixel in enumerate(decoded_data):
                    row = i // width
                    col = i % width
                    frame_data[row, col] = local_color_table[pixel]

                # Overlay the new frame_data onto the global frame
                global_frame_data[top_position:top_position+height, 
left_position:left_position+width] = frame_data

                frame_img = Image.fromarray(global_frame_data.astype('uint8'), 'RGB')
                frame_img.save(f'frame_{frame_counter}.png')

                frame_counter += 1

            elif block_type == b'\x21':  # Extension
                f.read(1)  # Extension function code
                extension_length = int.from_bytes(f.read(1), 'little')
                f.read(extension_length)  # Skip extension data
                while True:
                    sub_block_size = int.from_bytes(f.read(1), 'little')
                    if sub_block_size == 0:
                        break
                    f.read(sub_block_size)

read_and_dump_frames("giphy.gif.1")
```

For this we had to handle the LZW compression before writing to a image format. We can see that we successfully get frames out:



Trailer

The GIF trailer is a single-byte block containing the hexadecimal value `0x3B`. In ASCII, this corresponds to a semicolon (`;`).

Refactoring

At this point, I spent a bunch of time taking all of the code snippets above and making an overall python class. This brings them all together before diving into implementing the logic for hiding and recovering data.

Hiding Data

A common strategy used to hide data in images is called Least Significant Bit (LSB) steganography. In simple terms, it replaces the "least important" bits of the image with the data to be hidden.

```
def lsb_encode(self, frame_data, byte_array):
        for byte_index, byte_val in enumerate(byte_array):
            for bit_index in range(8):
                pixel_index = byte_index * 8 + bit_index
                frame_data[pixel_index] &= 0xFE  # Clear the least significant bit
                frame_data[pixel_index] |= (byte_val >> bit_index) & 1  # Set the
least significant bit
        return frame_data
```

The general logic for hiding data in a frame is as follows:

- Take the uncompressed frame data.
- Encode the hidden data (plus a magic signature to validate/detect the end later) into this uncompressed frame data.
- LZW compress the new frame data.
- Chunk the compressed data and generate the sub blocks containing the new frame data.
- Replace these sub blocks in the output GIF file

The relevant code snippet is shown below:

```
uncompressed_frame = self.lzw_decode(min_code_size, data)
        if self.hide:
            if self.frames < len(self.blobs) and len(self.blobs[self.frames] +
self.magic_code) > (len(uncompressed_frame) / 8):
                print("Warning: Blob to be hidden was too big, skipping")
                hidden_frame = uncompressed_frame
            elif self.frames < len(self.blobs):
                if len(self.blobs[self.frames]) > 0:
                    blob_to_hide = self.blobs[self.frames] + self.magic_code
                    hidden_frame = self.lsb_encode(uncompressed_frame, blob_to_hide)
                else:
                    hidden_frame = uncompressed_frame
            else:
                hidden_frame = uncompressed_frame
            hidden_frame_compressed = self.lzw_encode(min_code_size, hidden_frame)
            new_sub_blocks = self.generate_sub_blocks(hidden_frame_compressed)
            for block in new_sub_blocks:
                self.buffer.extend(block.sub_block_size.to_bytes(1, 'little'))
                self.buffer.extend(block.sub_block_data)
            self.append_read_to_buffer = True
```

Recovering Data

To recover data we just take the uncompressed frame data and run the following LSB decode function. I check for a magic signature as a way to validate things have gone to plan and to denote the end of the data we want.

```
def lsb_decode(self, frame_data):
        num_bytes = len(frame_data) // 8
        decoded_bytes = bytearray()
        for byte_index in range(num_bytes):
            decoded_byte = 0
            for bit_index in range(8):
                pixel_index = byte_index * 8 + bit_index
                decoded_byte |= (frame_data[pixel_index] & 1) << bit_index  # Extract
the least significant bit
            decoded_bytes.append(decoded_byte)
        if self.magic_code in decoded_bytes:
            return decoded_bytes.split(self.magic_code)[0]
        else:
            return bytearray()
```

GIFT Tool

We now have a Python library that implements all of the basic logic needed. Let's use a wrapper tool that makes it easy to play around with this stuff.

It has the following modes:

- hide - hide multiple files across multiple frames of a GIF
- recover - recover multiple files from multiple frames of a GIF
- spread - hide a single file across all of the frames of a GIF
- gather - recover a single file that is hidden across all frames of a GIF
- analyze - analyze a GIF and dump all the frames to PNG files

You can access the code at:
https://github.com/dtmsecurity/gift

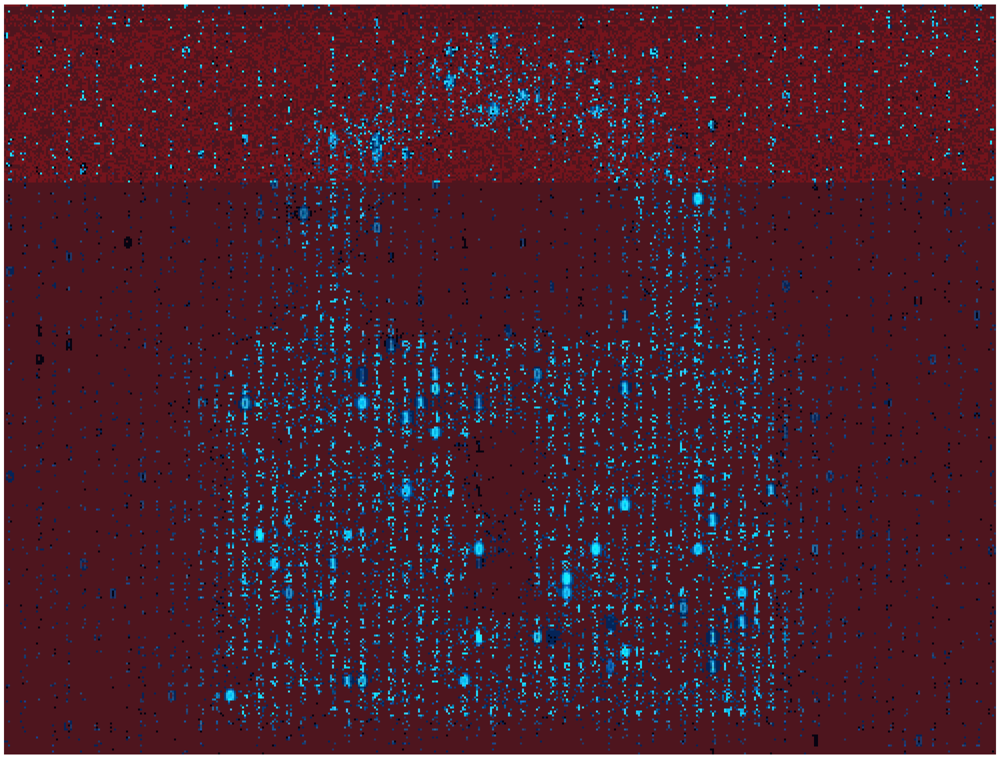
```
python3 gift-cli.py
usage: gift-cli.py [-h] [--source SOURCE] [--dest DEST]
{hide,recover,analyze,spread,gather} filenames [filenames ...]
gift-cli.py: error: the following arguments are required: mode, filenames
```

Let’s start playing - We are going to hide a text file and a jpg in a GIF:

```
python3 gift-cli.py --source giphy.gif --dest output.gif hide hello.txt meme.jpg
Hiding files in giphy.gif and writing to output.gif
We will hide: hello.txt
We will hide: meme.jpg
Doing magic...
Done...now writing to output.gif
```

Despite the payloads only being introduced to the first two frames, due to the nature of GIF files you can still see the artefacts of the encoding in the following frames. For this example, it’s pretty cool to visualise things at least. By choosing how much data to hide and targeting particular frames or spreading the data we can minimise any visible differences easily.

Below is the first two frames dumped, as you can see the small text file in Frame 1 is barely noticeable. The meme in frame 2 shows a bit more.



Let’s recover the files we hid:

```
python3 gift-cli.py --source output.gif recover recovered_hello.txt
recovered_meme.jpg
Recovering files from output.gif
Recovering recovered_hello.txt
Recovering recovered_meme.jpg

% shasum hello.txt recovered_hello.txt
22596363b3de40b06f981fb85d82312e8c0ed511  hello.txt
22596363b3de40b06f981fb85d82312e8c0ed511  recovered_hello.txt
% shasum meme.jpg recovered_meme.jpg
a1838d4cb7cd2311dae420ec6bc8688e56dc5414  meme.jpg
a1838d4cb7cd2311dae420ec6bc8688e56dc5414  recovered_meme.jpg
```

Awesome! We successful got the original files back. Now I wanted to be able to spread a single file across all frames to reduce the visual impact. For this we use the ‘spread’ feature of the tool.

```
python3 gift-cli.py --source giphy.gif --dest output.gif spread meme.jpg
Hiding file across frames of giphy.gif and writing to output.gif
We will hide: meme.jpg
We have split meme.jpg into 118
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 47
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Chunk of size 46
Doing magic...
Done...now writing to output.gif
```

You can now barely see the artefacts of encoding! Let’s use the ‘gather’ feature to recover our file.

```
python3 gift-cli.py --source output.gif gather recovered_meme.jpg
Recovering files from output.gif
Recovering recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 47 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg
Writing recovered blob of size 46 to recovered_meme.jpg

% shasum meme.jpg
a1838d4cb7cd2311dae420ec6bc8688e56dc5414  meme.jpg
% shasum recovered_meme.jpg
a1838d4cb7cd2311dae420ec6bc8688e56dc5414  recovered_meme.jpg
```

The final feature to introduce is the analyze function. This returns information on a GIF as well as dumping the frames:

```
python3 gift-cli.py analyze output.gif

---
GIF INFO
---
header = GIF89a
frames = 118
---
LOGICAL SCREEN DESCRIPTOR
---
screen_width = 500
screen_height = 375
packed_fields = 246
gct_flag = 1
color_res = 7
sort_flag = 0
gct_size = 6
bg_color_index = 57
pixel_aspect_ratio = 0
---
GLOBAL COLOR TABLE
---
gct_size = 128
gct_colors = [(21, 163, 221), (6, 3, 16), (15, 141, 200), (2, 101, 151), (56, 173,
230), (37, 202, 252), (5, 36, 87), (36, 217, 255), (39, 74, 122), (21, 202, 250),
(42, 118, 170), (2, 63, 124), (41, 100, 148), (1, 87, 151), (2, 104, 168), (1, 84,
135), (52, 202, 253), (2, 126, 185), (11, 124, 207), (2, 117, 171), (56, 217, 254),
(2, 70, 146), (31, 232, 255), (32, 56, 103), (23, 122, 185), (24, 84, 134), (22, 103,
162), (22, 87, 151), (20, 118, 169), (21, 185, 241), (21, 68, 145), (37, 183, 242),
(2, 184, 240), (49, 143, 197), (55, 190, 244), (83, 38, 69), (28, 87, 165), (10, 178,
226), (38, 185, 221), (37, 212, 243), (52, 210, 241), (71, 200, 252), (20, 214, 245),
(12, 237, 255), (21, 218, 255), (5, 104, 186), (21, 103, 188), (0, 214, 244), (6, 86,
171), (3, 202, 247), (5, 219, 254), (66, 100, 140), (72, 192, 187), (12, 248, 230),
(49, 224, 210), (122, 164, 195), (78, 21, 30), (116, 20, 28), (80, 4, 8), (108, 34,
44), (146, 20, 30), (145, 34, 42), (153, 9, 15), (178, 20, 23), (196, 17, 26), (115,
61, 100), (107, 88, 134), (30, 205, 151), (42, 161, 75), (109, 84, 32), (207, 46,
47), (146, 142, 67), (212, 71, 74), (255, 255, 255), (0, 0, 0), (0, 0, 0), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0,
0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0,
0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0,
0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0,
0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0)]
---
APPLICATION EXTENSIONS
---
block_size = 11
app_identifier = NETSCAPE
app_auth_code = 322e30
sub_block_size: 3
sub_block_data: b'\x01\x00\x00'
```

```
sub_block_size: 60
sub_block_data: b'?xpacket begin="\xef\xbb\xbf" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpm'
sub_block_size: 101
sub_block_data: b'ta xmlns:x="adobe:ns:meta/" x:xmptk="Adobe XMP Core 6.0-c002
79.164460, 2020/05/12-16:04:17        ">'
sub_block_size: 32
sub_block_data: b'<rdf:RDF xmlns:rdf="http://www.w'
sub_block_size: 51
sub_block_data: b'.org/1999/02/22-rdf-syntax-ns#"> <rdf:Description r'
sub_block_size: 100
sub_block_data: b'f:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/"
xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/" xm'
sub_block_size: 108
sub_block_data: b'ns:stRef="http://ns.adobe.com/xap/1.0/sType/ResourceRef#"
xmp:CreatorTool="Adobe Photoshop 21.2 (Macintosh)"'
sub_block_size: 32
sub_block_data: b'xmpMM:InstanceID="xmp.iid:2A4A11'
sub_block_size: 55
sub_block_data: b'951C011EB8B3DEEE1F100462C" xmpMM:DocumentID="xmp.did:2A'
sub_block_size: 52
sub_block_data: b'A117A51C011EB8B3DEEE1F100462C"> <xmpMM:DerivedFrom s'
sub_block_size: 116
sub_block_data: b'Ref:instanceID="xmp.iid:2A4A117751C011EB8B3DEEE1F100462C"
stRef:documentID="xmp.did:2A4A117851C011EB8B3DEEE1F100462C'
sub_block_size: 34
sub_block_data: b'/> </rdf:Description> </rdf:RDF> <'
sub_block_size: 47
sub_block_data: b'x:xmpmeta> <?xpacket end="r"?
>\x01\xff\xfe\xfd\xfc\xfb\xfa\xf9\xf8\xf7\xf6\xf5\xf4\xf3\xf2\xf1\xf0'
sub_block_size: 239
sub_block_data:
b'\xee\xed\xec\xeb\xea\xe9\xe8\xe7\xe6\xe5\xe4\xe3\xe2\xe1\xe0\xdf\xde\xdd\xdc\xdb\xd
a\xd9\xd8\xd7\xd6\xd5\xd4\xd3\xd2\xd1\xd0\xcf\xce\xcd\xcc\xcb\xca\xc9\xc8\xc7\xc6\xc5
\xc4\xc3\xc2\xc1\xc0\xbf\xbe\xbd\xbc\xbb\xba\xb9\xb8\xb7\xb6\xb5\xb4\xb3\xb2\xb1\xb0\
xaf\xae\xad\xac\xab\xaa\xa9\xa8\xa7\xa6\xa5\xa4\xa3\xa2\xa1\xa0\x9f\x9e\x9d\x9c\x9b\x
9a\x99\x98\x97\x96\x95\x94\x93\x92\x91\x90\x8f\x8e\x8d\x8c\x8b\x8a\x89\x88\x87\x86\x8
5\x84\x83\x82\x81\x80\x7f~}|{zyxwvutsrqponmlkjihgfedcba`_^]\\
[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;:9876543210/.-,+*)(\'&%$#"!
\x1f\x1e\x1d\x1c\x1b\x1a\x19\x18\x17\x16\x15\x14\x13\x12\x11\x10\x0f\x0e\r\x0c\x0b\n\
t\x08\x07\x06\x05\x04\x03\x02\x01\x00'
---snip---
IMAGE DESCRIPTORS
---
left_position = 0
top_position = 0
width = 500
height = 375
local_color_table_flag = 0
interlace_flag = 0
sort_flag = 0
reserved = 0
```

```
local_color_table_size = 0
local_color_table = [(21, 163, 221), (6, 3, 16), (15, 141, 200), (2, 101, 151), (56,
173, 230), (37, 202, 252), (5, 36, 87), (36, 217, 255), (39, 74, 122), (21, 202,
250), (42, 118, 170), (2, 63, 124), (41, 100, 148), (1, 87, 151), (2, 104, 168), (1,
84, 135), (52, 202, 253), (2, 126, 185), (11, 124, 207), (2, 117, 171), (56, 217,
254), (2, 70, 146), (31, 232, 255), (32, 56, 103), (23, 122, 185), (24, 84, 134),
(22, 103, 162), (22, 87, 151), (20, 118, 169), (21, 185, 241), (21, 68, 145), (37,
183, 242), (2, 184, 240), (49, 143, 197), (55, 190, 244), (83, 38, 69), (28, 87,
165), (10, 178, 226), (38, 185, 221), (37, 212, 243), (52, 210, 241), (71, 200, 252),
(20, 214, 245), (12, 237, 255), (21, 218, 255), (5, 104, 186), (21, 103, 188), (0,
214, 244), (6, 86, 171), (3, 202, 247), (5, 219, 254), (66, 100, 140), (72, 192,
187), (12, 248, 230), (49, 224, 210), (122, 164, 195), (78, 21, 30), (116, 20, 28),
(80, 4, 8), (108, 34, 44), (146, 20, 30), (145, 34, 42), (153, 9, 15), (178, 20, 23),
(196, 17, 26), (115, 61, 100), (107, 88, 134), (30, 205, 151), (42, 161, 75), (109,
84, 32), (207, 46, 47), (146, 142, 67), (212, 71, 74), (255, 255, 255), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0,
0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0,
0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0,
0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0,
0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0),
(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0)]
---snip---
DUMP FRAMES
---
writing frame_0.png
writing frame_1.png
writing frame_2.png
writing frame_3.png
writing frame_4.png
writing frame_5.png
writing frame_6.png
writing frame_7.png
writing frame_8.png
writing frame_9.png
writing frame_10.png
---snip---
```