1 Project 1

Due: Feb 14 by 11:59p

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. This is followed by an example which should help understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To get you to write a simple but non-trivial JavaScript program.
- To make you understand the representation of unsigned integers/characters and the use of bit twiddling.
- To allow you to familiarize yourself with the programming environment you will be using in this course.
- To introduce you to steganography.

1.2 Background

Ultimately, all data in computers is represented using binary numbers. Various mappings (known as encoding schemes) are used to map high-level data like a string, number or instance of an object class to the binary numbers within a computer.

Strings represent a sequence of characters. Each character is represented using some encoding like ASCII or UTF-8.

For example, the ASCII encoding for the 5-character string `"hello"` would be:

```
0x68 'h', 0x65 'e', 0x6c 'l', 0x6c 'l', 0x6f 'o'
```

It is important to emphasize that all data are merely sequences of bytes. A particular sequence of bytes like those above, when associated with an encoding like ASCII can be interpreted as a textual string. OTOH, the same sequence of bytes could be interpreted as the binary data bytes of an image or the instructions of a program. So the meaning of a sequence of data bytes depends on how they are being interpreted in the current context.

Steganography is the art or practice of concealing a message, image, or file within another message, image, or file. A simple example of steganography in the physical world is using *invisible ink* to conceal a secret message in a normal-looking letter. A historical use was by a US POW *blinking out* a secret message using Morse code when forced to participate in a propaganda video.

In this project, we will use bit-twiddling to conceal a string message within a PPM image file. The inefficient PPM format was chosen over more popular and practical formats like GIF or PNG as it is extremely simple to understand and manipulate.

The PPM format allows easy steganography as random changing of less-significant bits do not have any easily visible effect on the displayed image. For example, here is a image

and here is the same image containing the hidden message `"hello"`.

[As most browsers do not display `.ppm` images, the images shown in this document are `.png` versions of the `.ppm` images].

1.3 Requirements

Submit a `submit/prj1` directory on your gitlab repository with contents set up such that it contains a `steg.js` file. It should be possible to run `steg.js` with upto two arguments:

- When invoked with zero arguments, `steg.js` should simply output a usage message on standard error.
- When invoked with one argument, `steg.js` should unhide the message concealed in the PPM file named by its first argument and print it on standard output followed by a newline character.
- When invoked with two arguments, `steg.js` should hide the message specified by its second argument in the PPM image file named by its first argument and write the contents of the resulting image on standard ouput.

The program should detect errors in its inputs. To facilitate grading, all error messages must start with the corresponding error ID.

To ensure that source code is well-documented while avoiding inconsistencies with this document, the detailed specifications are given only in the code documentation in `steg_module.js`.

- The documentation for the `hide()` method describes the method which **must** be used for hiding messages within images as well as the error conditions which must be detected.
- The documentation for the `unhide()` method describes the error conditions which must be detected when recovering hidden messages.

1.4 Example Log

The behavior of the `steg` program is illustrated by the following annotated log:

```
$ ./steg.js
usage: steg.js PPM_FILE_NAME [MSG]
# hide "hello world" message in t.ppm
$ ./steg.js ~/cs580w/projects/prj1/aux/in/rose.ppm 'hello world' >t.ppm
# unhide message from t.ppm
$ ./steg.js t.ppm
hello world
# attempt to hide message in a file which already contains a message
$ ./steg.js t.ppm 'goodbye'
STEG_MSG: t.ppm: image already contains a hidden message
# attempt to unhide from a file without a message
$ ./steg.js ~/cs580w/projects/prj1/aux/in/rose.ppm
STEG_NO_MSG: rose.ppm: image does not have a message
# output image parameters
$ identify ~/cs580w/projects/prj1/aux/in/rose.ppm
...rose.ppm PPM 70x46 70x46+0+0 8-bit sRGB 9.67KB 0.000u 0:00.000
#maxMsgSize = width*height*nBytesPerPixel/bitsPerChar - nStegMagicBytes
$ echo $((70*46*3/8 - 3))
1204
# hide a message of maxMsgLen = 1204 - 1 'x's.
$ ./steg.js ~/cs580w/projects/prj1/aux/in/rose.ppm \
    `printf "%1203s" | tr ' ' x` >t-xs.ppm
#Extract it.
$ ./steg.js t-xs.ppm
xxxx...xxx
#Verify size (count includes newline after extracted message; hence 1204)
$ ./steg.js t-xs.ppm | wc -c
1204
# compute # of pixel bytes
$ echo $((70*46*3))
9660
# show size of image file; so header occupies 9673 - 9660 = 13 bytes
$ wc -c t-xs.ppm
9673 t-xs.ppm
# compute offset of last pixel byte (-1 to get offset)
```

```
$ echo $((13 + 70*46*3/8*8 - 1))
9668
# clobber byte at that offset to be odd; hence clobbering NUL-terminator
$ echo '1' | dd of=t-xs.ppm bs=1 seek=9668 count=1 conv=notrunc
...
1 byte copied, 0.000113903 s, 8.8 kB/s
# now try to unhide: message terminator not found
$ ./steg.js t-xs.ppm
STEG_BAD_MSG: t-xs.ppm: bad message
# hiding a message with just one more character is too much
$ ./steg.js ~/cs580w/projects/prj1/aux/in/rose.ppm \
    `printf "%1204s" | tr ' ' x` >t-xs.ppm
STEG_TOO_BIG: rose.ppm: message too big to be hidden in image
$
```

1.5 Understanding an Image with a Hidden Message

The image contained in `aux/out/rose-hello.ppm` contains the hidden message `"hello"`. Let's dump out the contents of the initial portion of this file and try to understand how the message is concealed within it:

```
$ od -N 128 -t x1 -c ~/cs580w/projects/prj1/aux/out/rose-hello.ppm
0000000  50  36  0a  37  30  20  34  36  0a  32  35  35  0a  30  2f  2d
          P   6  \n   7   0       4   6  \n   2   5   5  \n   0   /   -
0000020  33  30  2e  37  33  2e  39  33  2f  3a  33  2c  38  32  2d  39
          3   0   .   7   3   .   9   3   /   :   3   ,   8   2   -   9
0000040  30  2c  39  31  2f  38  31  2d  38  31  2c  36  2e  2c  35  2d
          0   ,   9   1   /   8   1   -   8   1   ,   6   .   ,   5   -
0000060  2a  34  2d  28  35  2c  2b  35  2c  2b  31  2c  26  30  2f  27
          *   4   -   (   5   ,   +   5   ,   +   1   ,   &   0   /   '
0000100  34  31  2b  36  34  2c  39  37  2e  3f  3b  2f  47  3e  32  4a
          4   1   +   6   4   ,   9   7   .   ?   ;   /   G   >   2   J
0000120  42  34  4c  40  32  4e  42  32  55  41  32  74  44  34  9a  43
          B   4   L   @   2   N   B   2   U   A   2   t   D   4 232   C
0000140  33  b4  41  35  c5  45  3d  e0  44  47  ed  43  46  f6  3d  42
          3 264   A   5 305   E   = 340   D   G 355   C   F 366   =   B
0000160  f1  3c  40  d6  40  3b  b8  3f  2f  b2  3f  2d  ad  40  2d  a3
        361   <   @ 326   @   ; 270   ?   / 262   ?   - 255   @   - 243
0000200
$
```

The dump contains the first 128 bytes of the file, 16 bytes per line. Each byte is dumped out in both hex as well as characters (when it corresponds to a ASCII character).

Looking at the characters, we clearly see the image header:

```
P6
70 46
255
```

The pixel data start with the byte with value `0x30` after the `\n` after the 255.

Extracting the message stored in the pixel bytes is easy: starting with the first pixel byte, segment the stream of data bytes into groups of 8 and then extract the message bits from the LSB of each byte; if the value of the byte is even, the the message bit is 0; if it is odd then it is 1. So we have:

Pixel Bytes								Binary	Hex	Char
30	2f	2d	33	30	2e	37	33	0111_0011	0x73	's'
2e	39	33	2f	3a	33	2c	38	0111_0100	0x74	't'
32	2d	39	30	2c	39	31	2f	0110_0111	0x67	'g'
38	31	2d	38	31	2c	36	2e	0110_1000	0x68	'h'
2c	35	2d	2a	34	2d	28	35	0110_0101	0x65	'e'
2c	2b	35	2c	2b	31	2c	26	0110_1100	0x6c	'l'
30	2f	27	34	31	2b	36	34	0110_1100	0x6c	'l'
2c	39	37	2e	3f	3b	2f	47	0110_1111	0x6f	'o'
3e	32	4a	42	34	4c	40	32	0000_0000	0x00	'\0'

Hence the image contains the string `"stghello"` followed by a `NUL`-terminator. This is exactly the `"hello"` message concatenated to `STEG_MAGIC`.

1.6 Provided Files

The prj1 directory contains a start for your project. It contains the following files:

steg.js A file which is used as an entry point for the program. It implements the required command-line processing and calls the appropriate functions in `steg_module.js`. You should not need to change this file.

steg_module.js This file contains a constructor and empty versions of the functions (with dummy return values) you need to implement for this project. You may add any auxiliary functions you need to this file.

ppm.js This file contains a constructor to build a `Ppm` object which represents an abstraction of a PPM image. It can be called with a `Uint8Array` array of bytes, or by an existing `Ppm` object. In the latter case, it returns a copy of its argument. The `pixelBytes` property of the `Ppm` object provides the byte array where messages should be hidden.

README A README file which must be submitted along with the project. It contains a initial header which you must complete (replace the dummy

entries with your name, B-number and email address at which you would like to receive project-related email). Following a empty line after the header you may provide any content which you would like read during the grading of your project.

The `aux` directory contains auxiliary files:

- Images which you can use for testing. Note that each `*.ppm` image, has a corresponding `*.png` image as the former do not display in a browser whereas the latter do. Note that it is possible to use Image Magick's convert program to round-trip between `ppm` and `png` without loosing hidden messages.

1.7 Hints

The following points are not prescriptive in that you may choose to ignore them as long as you meet all the project requirements.

You will need to look at binary data. You can use `od` as in the example above. You can also use emacs (when you display a image file within emacs, you can use `^C^C` to see the binary data).

The provided files take care of all command-line processing and the reading and parsing of PPM image files. All you need to do is implement the `hide()` and `unhide()` methods for the `StegModule` object.

You may proceed as follows:

1. Get a general understanding of the principles of steganography and PPM image files by reading the beginnings of the corresponding Wikipedia articles or any other WWW resources.
2. Read and understand the bit-manipulation operations available in JavaScript: `&`, `|`, `~`, `<<` and `>>` and how they can be used.

   You will be using bit manipulation for 2 things:

   (a) Setting or extracting the LSB of a pixel data byte. You can set the LSB of `pixel` to 1 by using a bit-wise or: `pixel = pixel | 1;`. You can set the LSB of `pixel` to 0 by using a bit-wise and: `pixel = pixel & ~1;`, You can extract the LSB of a pixel byte using simply `pixel & 1`.

   (b) For processing the bits within a message character, use a `mask` set up for the MSB by initializing it to `1u << (8 - 1)` (we are assuming that the number of bits per character is 8). You can then loop through all the bits in a message char by right-shifting `mask` on each loop iteration, terminating the loop when `mask` becomes 0. On each loop iteration you can use `mask` to extract or set the corresponding message bit.

3. To get started on actually implementing your project, first ensure that you have set up your VM as described in the *Git Setup* document. Once that is done, simply copy the `prj1` directory into your work directory.

4. Implement the `unhide()` method first. You can do so before implementing the `hide()` method, as you can test your `unhide()` implementation by attempting to use it to extract the `"hello"` message hidden in rose-hello.ppm.

   Note that you have 2 kinds of uses for extracting a message:

   (a) The calls to `unhide()` from a client. Here you will be returning the message without the `STEG_MAGIC` prefix.

   (b) Internal calls from your own code to extract `STEG_MAGIC` to check for `STEG_MSG` / `STEG_NO_MSG` errors. You will need to check this for both your `hide()` and `unhide()` methods.

   Write a lower-level routine which accomodates both these uses. Then the above 2 uses can simply be wrappers around this low-level routine.

   Note that you can convert a character code `charCode` to a character using `String.fromCharCode(charCode)`.

5. Now implement the `hide()` method. You can test your implementation using the example covered earlier. Note that you can get the character code of character `c` using `c.charCodeAt()`.

6. Iterate until you meet all requirements. Verify that you detect errors and return appropriate error messages.