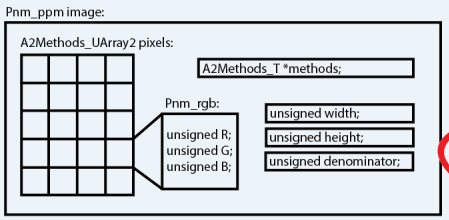
COMP40 HW4 Design Doc
Sitara Rao (srao03) and Era Iyer (eiyer01)
10/15/2019

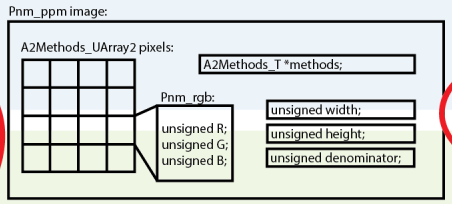
Original Image

Module 1: ppm_mod.c

1.a) Read in ppm file
Input: ppm image from stdin/ command line filename
Output: Pnm_ppm object



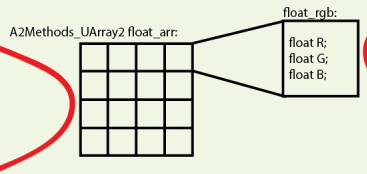
1.b) Trim to even width and height
Input: Pnm_ppm object
Output: Pnm_ppm object



8. Write Pnm_ppm to stdout
Input: Pnm_ppm object
Output: Ppm image in binary

Module 2: float_mod.c

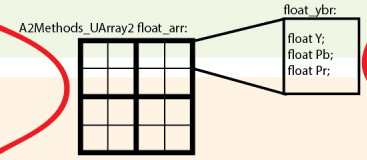
2.a) Convert RGB unsigned ints to RGB floats
Input: Pnm_ppm object
Output: A2Methods_UArray2 object, where each elem is a struct of 3 floats (struct float_rgb)



2.b) Convert floats R,G,B to floats Y, Pb, Pr
Input: A2Methods_UArray2 object, where each elem is a struct of 3 floats (struct float_rgb)
Output: A2Methods_UArray2 with blocksize = 2, where each elem is a struct of 3 floats (struct float_ybr)

7.b) Choose a denominator to scale R, G, B back to unsigned ints
Input: A2Methods_UArray2 object, where each elem is a struct float_rgb
Output: Pnm_ppm object

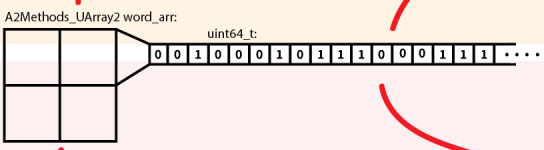
7.a) Convert floats Y, Pb, Pr to floats R, G, B
Input: A2Methods_UArray2 object, where each elem is a struct of 3 floats (struct float_ybr)
Output: A2Methods_UArray2 where each elem is a struct_rgb



Module 3: word_mod.c

3. Calculate a, b, c, d, avg(Pb), avg(Pr) for each block
Input: A2Methods_UArray2 with blocksize = 2, where each elem is a struct of 3 floats (struct float_ybr)
Output: A2Methods_UArray2 object, half the height and half the width of the input. Each elem is a uint64_t that represents a block of the input

6. Reconstruct Y1, Y2, Y3, Y4 (for each cell) and Pb, Pr (for the block)
Input: A2Methods_UArray2 with each elem a struct word32bit
Output: A2Methods_UArray2 object, twice the height and twice the width of the input. Each elem is a float_ybr; Y is unique for each cell, while Pb and Pr are the same across each 4-cell block



Module 4: bit_mod.c

4. Print compressed image to stdout
Input: A2Methods_UArray2 object containing uint64_t objects
Output: a header, followed by a row-major printing of 32 bit binary sequences (the contents of each uint64_t in the array) in big-endian order

5. Read 32-bit sequences in from stdin
Input: a 2 line header followed by a sequence of 32-bit words
Output: A2methods_Uarray2 with height/2 and width/2 dimensions (given in header), each elem is a uint64_t containing a word from input

Compressed Image

More details:

- There are 4 large steps to our compression / decompression scheme, separated into 4 color-coded modules in the diagram above. Each part of step 1 will be taken care of by a function defined in ppm_mod.c (tentative name), and so on for modules 1-4.
- We have organized modules so that their common "secret" is the transformation of one underlying data structure to another. For example, during compression a Pnm_ppm goes into module 1 and an A2Methods_UArray2 of float_ybr structs comes out (and vice versa for decompression). The result is a fairly modular program; for example, modules 1, 2, and 4 have no knowledge of bitpacking and rely on none of the code in bitpack.c, because that whole process happens in module 3.

Module 1: Here we anticipate losing info if our input image has uneven height or width.

- If both dimensions are even, no info lost
- If not, bottom row and/or rightmost column will be discarded during compression. This information cannot be re-encoded into the image during decompression.

Module 2: Again, information *could* be lost here.

- Compression: no info lost. Chroma values produced are arithmetical transformations of the r/g/b : denominator ratio, and can be reverse calculated to yield the original ratios.
- In decompression, when we choose a denominator and scale the rgb ratios back to unsigned ints, we could potentially lose image quality. For example, if the original denominator is 255 and we choose to re-scale to denominator = 2 during decompression, there will only be 3 possible values for each pixel numerator (0, 1, 2) and the image will lose depth compared to original. Again this may or may not result in information loss, depending on the original denominator before compression and the denominator we scale to during decompression.

Module 3: Where quantization and bitpacking happen; we anticipate losing info.

- The amount of info lost in quantization will once again depend on the original image, specifically the degree of variation in color. When we get quantized chroma values for a block, we are measuring average saturation as well as how much the color varies from top→bottom, left→right, and along diagonals.
- According to the spec, since we are not quantizing linearly, this will result in highly saturated test images losing more info during this step than less-saturated images. This is a trade-off because it gives us more precision in the "normal" use case, where we have less saturation and indices P_B and P_R clustered closer to 0.
- We do not anticipate losing any information in bitpacking, because we will be able to fit the block's quantized information into one 32-bit word.

Module 4: No info lost

- Module 4 is simply printing information we have already processed (in compression) or reading info from input and storing it in memory (decompression).

Implementation/Testing Plan:
We plan to test the forward and reverse steps of each module at a time, first compressing and then decompressing the image in order to ensure that the image returned looks reasonable. We will also use our working ppmdiff program to find the mean root square difference in the pixel values of the original and output images to get a numerical value for the degree of difference between the two. Specifically, we will proceed in the order that follows:

1.a) Read in ppm file

- Print out known information about the pnm_ppm object created.
    - This includes the height and width since we know the dimensions of the original image that was read in.

1.b) Trim to even width and height

- Provide an image that needs to be trimmed and printing out the height and width of the image before and after trimming.
    - The output we would expect is that any image with an odd height or width will be reduced by 1,
    - The output of the image with an even height or width should remain the same.

8. Write Pnm_ppm to stdout

- Call Pnm_ppmwrite on the Pnm_ppm object produced as a result of step 1.b).
- Pipe the output to a file and then call ppmdiff on the original ppm file and the piped output.
    - The only changes made thus far to the original image will be a row and/or column that gets trimmed off, if anything, and the root mean square difference when running ppmdiff should thus be very small or 0.

2.a) Convert RGB unsigned ints to RGB floats

- Print out the unsigned values from the Pnm_rgb struct and denominator from Pnm_ppm object alongside the float values from our float_rgb struct.
    - Ensures that the values we are putting into the float_rgb structs are what we want.
- After creating the A2Methods_UArray2 object that holds our float_rgb structs, we can print out information about this A2Methods_UArray2 object.
    - The dimensions of this UArray2 should be the same as the Pnm_ppm's A2Methods_UArray2 dimensions.

7.b) Choose a denominator to scale RGB floats back to unsigned ints

This step will only be written after completing and testing step 8. Once we know that step 8 works properly…

- Pass the pnm_ppm output of this step back to step 8.
- Pipe the output into another file and change the value of the denominator in order to ensure that the image outputted changes as the size of the denominator changes

    - We expect less color resolution for small denominators

2.b) Convert RGB floats to floats Y, P_B, P_R

- Print out values from our float_rgb struct next to the values we calculate and store in our float_ybr struct.
    - These values should be the same as what we expect from doing the mathematical transformations out by hand.
- Also verify that the dimensions of the A2Methods_UArray2 are the same as those of the one passed into this step.

7.a) Convert floats Y, P_B, P_R to floats R, G, B

- Once we have transformed our A2Methods_UArray2 of float_ybrs to one of float_rgbs, we will pass the resulting array back to step 7.b) which goes back to step 8.
    - Resulting image will be compared to the original to verify there are no large discrepancies.

3. Calculate a, b, c, d, average P_B and average P_R for each 4-cell block

- Print out the dimensions of the new A2Methods_UArray2 object that is returned
    - Make sure dimensions are half of the A2Methods_UArray2 object that was passed in.

6. Calculate Y_1, Y_2, Y_3, Y_4, and P_B, and P_R for each block

- Check that the array produced is 2x the height and width of input array from step 3.
- Print out Y, P_B, and P_R values at various blocks in the array.
- Verify that Y is unique for every cell in the block, while P_B and P_R are the same across each 4-cell block.
- Lastly we will pass the resulting array back to step 7.a) and compare output and original images visually and using ppmdiff.

4. Print compressed image to stdout

- Verify that the header printed to stdout has the same height and width as the original image.
- Print out first uint64_t from array that was input
    - Make sure it corresponds to the first printed word and that it is done so in big-endian order.
    - Do the same for various other uint64_t elements in the array.

5. Read in 32-bit sequences from stdin

- We will first compress an image using steps 1-4, then pipe that output to a file.
- Pipe the file into this step, and pass the array of uint64_ts produced by this step back to step 6
    - Compare output and original images visually and using ppmdiff.