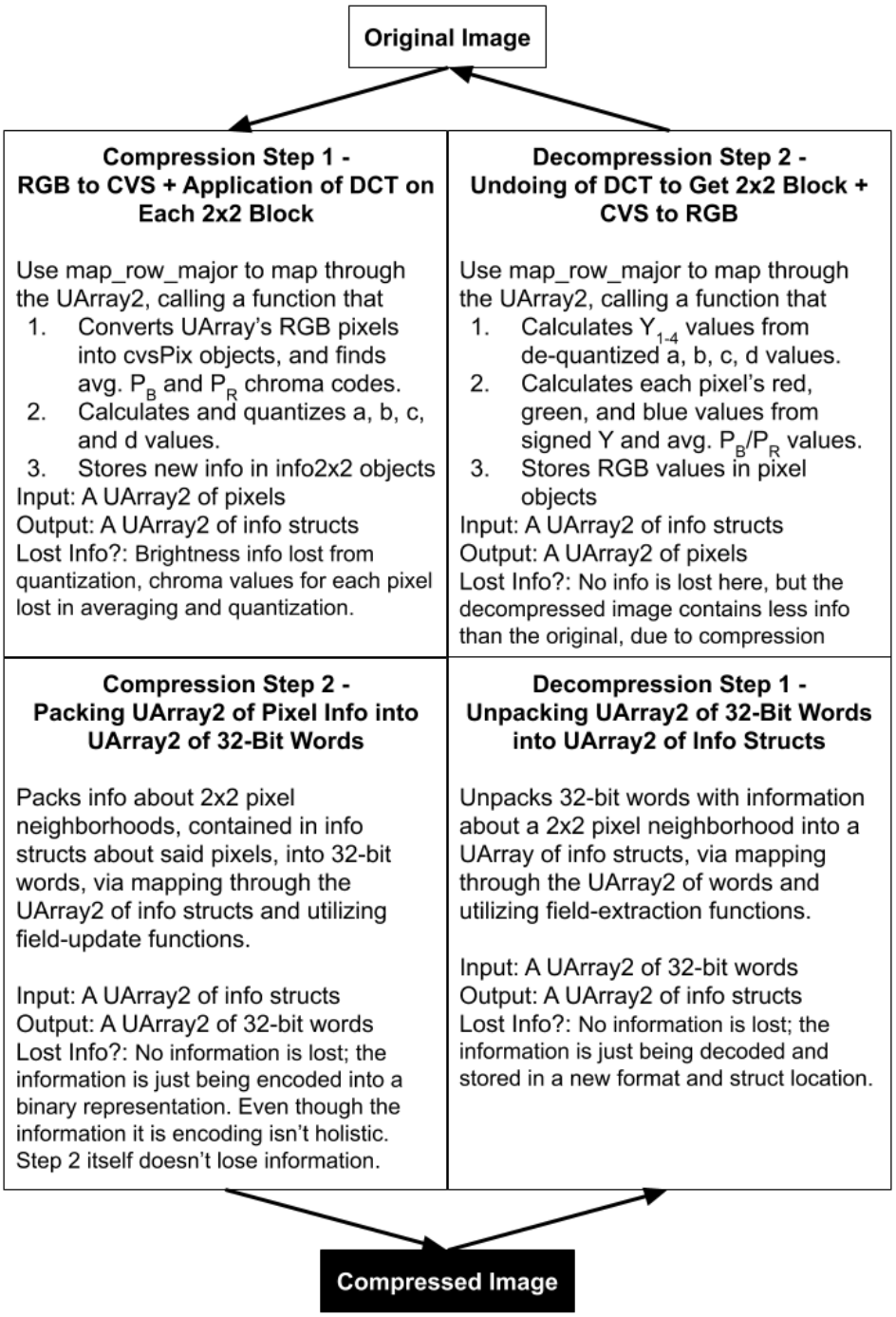
John Stewart and Lewis Bobrow
Monday, February 24th 2020
Comp40

HW4 DESIGN DOCUMENT



KEY

- CVS = Component Video Color Space
- DCT = Discrete Cosine Transformation
- cvsPix = A struct which stores Y, P_B, and P_R values of each pixel
- info2x2 = A struct which stores a, b, c, d, and avg. P_B/P_R values of each 2x2 pixel neighborhood

STRUCTS THAT WE USE

- A regular RGB **pixel struct** (contains red, green, and blue values for each pixel)
- **cvsPix:** A CVS color space **pixel struct** (contains Y, P_B, and P_R values for each pixel)
- **info2x2:** An 2x2-block **information struct** (contains the chroma codes for the avg $\overline{P_B}$ / $\overline{P_R}$ values, as well as cosine coefficients, for a block of each block of four pixels)

IN-DEPTH DESCRIPTIONS

COMPRESSION (before steps, read in PPM image and trim rows and columns, store inform)

1. **Step 1** - RGB to CVS and Application of DCT on Each 2x2 Block
    a. What is accomplished?
        i. Map through the UArray2 of pixels
            1. Create 4 new cvsPix objects and a new code word
            2. Assign the values of the cvsPix objects using their respective equations and the RGB values for each pixel object, then input the cvsPix objects into the UArray2 (use RGBandCVS())
            3. Get averages for $\overline{P_B}$ and $\overline{P_R}$ using each cvsPix's P_B and P_R values, and make each four-bit chroma code (call index_to_chroma with the equation to calculate the sum of P_B and P_R
            4. Calculate a, b, c, and d using the $Y_{1\text{-}4}$ values (use LumaAndCos())
            5. Quantize b, c, and d based on the 31-element array and their matching quantization values, quantize a by multiplying it by 511 and then rounding
    b. Input: 4 regular RGB pixels
    c. Output: An info2x2 struct with $\overline{P_B}$, $\overline{P_R}$, a, b, c, d
    d. Whether information is lost and why?
        i. Brightness information would be pretty much retained, but it wouldn't be exacted as a result of quantization
        ii. Chroma values for individual pixels would be lost, because P_B and P_R represent averages and are themselves quantized, necessitating the loss of precise chroma information for each pixel
2. **Step 2** - Packing uarray2 of pixel info into uarray2 of 32-bit words
    a. What is accomplished?
        i. Packing differently-sized small fields containing info about a 2x2 pixel neighborhood into one 32-bit word
        ii. Map through the UArray2 of info2x2 structs
            1. Use the field-update functions to store the values from each info2x2 struct (a, b, c, d, P_b, P_r) in a 32-bit word
            2. Store this word in the corresponding location in a UArray2 of words (passed in as the closure variable)
    b. Input: A uarray2 of info2x2 structs
    c. Output: A uarray2 of 32-bit code words
    d. Whether information is lost and why?

i. No information is lost, because this packing step is simply encoding the information defined in Step 1 into a new binary representation. The cosine coefficient values are factors of 2 a negative/positive delineation, so they can be represented entirely as bits without losing info, while the P_B and P_R

DECOMPRESSION

1. **Step 1** - Appropriate Name (Bit Unpacking)
    a. What is accomplished?
        i. Unpacking 32-bit words containing info about a 2x2 pixel neighborhood into a UArray2 of info2x2 structs (a, b, c, d, $\overline{P_B}$, $\overline{P_R}$)
        ii. Map through the UArray2 of words
            1. Use the field-extraction functions to extract the encoded information defined above from the 32-bit word
            2. Store the information in the corresponding location in a UArray2 of info2x2 structs (passed in as the closure variable)
    b. Input: A uarray2 of 32-bit code words
    c. Output: A uarray2 of info2x2 structs with $\overline{P_B}$, $\overline{P_R}$, a, b, c, d
    d. Whether information is lost and why?
        i. No information is lost, because we are simply taking the information from the word and storing it in a new format and struct.
2. **Step 2** - Undoing DCT to get a 2x2 CVS then converting to RGB
    a. What is accomplished?
        i. Create and map through the UArray2 of info2x2 structs, with width and height defined by reading in a header file.
            1. Create 4 new RGB pixels and 4 new cvsPix objects
            2. Assign b, c, and d to the exact dequantized values associated with each element in the 31-element unsigned array. Divide a by 511 (use LumaAndCos())
            3. Calculate $Y_{1\text{-}4}$ values from a, b, c, and d using the given equations (separate calculation function)
            4. Call chroma_of_index to generate signed $\overline{P_B}$ and $\overline{P_R}$ values, which you will use when getting RGB values. Assign the P_B, P_R, and Y values in each of the cvsPix objects.
            5. Get the red, green, and blue values for each pixel using the $\overline{P_B}$ and $\overline{P_R}$ values, along with the $Y_{1\text{-}4}$ values and the denominator we chose when creating the 2D array, and then calculate Assign them to their respective RGB pixel. (use RGBandCVS())
    b. Input: An info2x2 struct, containing $\overline{P_B}$, $\overline{P_R}$, a, b, c, d
    c. Output: 4 new RGB pixels (compile into the 2D array and print out)
    d. Whether information is lost and why?

i. No information is lost, but the resulting decompressed image will contain less information than the original image (in particular the chroma values) as a result of the compression step

INTENDED MODULARITY

RBGandCVS(): Converts RGB color space pixels to component video space cvs pix objects (and vice versa) Utilize equations in the spec and a boolean integer variable to denote whether to switch from RGB to component-video or component-video to RGB
LumaAndCos(): Converts Y values (luma values) to cosine coefficients (and vice versa) Utilizes equations in the spec and a boolean integer variable to denote whether to switch from luma values to cosine coefficients or the other way around
Mapping Functionality: We will be utilizing a mapping function in order to convert all of the pixels/structs in a certain UArray2 into new objects/structs, as well as to calculate various amounts.
Width and LSB Storage (for returning information): We would also create a function that would return a struct of widths and least significant bits for each bit of information stored in a 32-bit word. Then if the format of the word is changed, the function's parameters can be updated, and new LSB's would be returned.
Mask Generation (for updating information): We would create a function that would return a struct of custom masks for each bit of information stored in a 32-bit word, based on the least significant bit for each bitset of information. Then, if the format of the word is changed, we would just need to call the function to generate custom masks for the information we are trying to access, without having to update a bunch of lines to create new masks. The modularity of this function would also be strengthened by the above Width and LSB storage function, as it would use the width and least significant bit fields returned from that function as parameters.

TESTING (FOR EACH STEP OF IMPLEMENTATION):

1. General Note: For testing compression and decompression, we would be utilizing a small group of images (in their various forms and formats). We would have one 2x2 ppm file, one slightly larger ppm file, a medium-sized ppm file, and a very large file; the single 2x2 ppm file and the small ppm file would be used for testing correctness, while the other two would be used for testing general functionality and high memory values.
2. **Compression**
    a. **Step 1:** We would test small for correctness, and test large for completion. We would create a test file for our Step 1 functionality, which would provide a 2-pixel-by-2-pixel PPM file and declare a set of variables representing the correct amount for each element in the info2x2 struct

output. We can calculate the correct values of a, b, c, d, $\overline{P_B}$, $\overline{P_R}$ manually and then printf the values in each the info2x2 struct to ensure that they match. This way, we can check that the functionality works correctly, before using larger PPMs in the test file to make sure the functionality can handle them, in terms of memory and time. We would also test individual functionalities as we implement them, adding cases into the test file to test these parts before moving on with coding the implementation; these functionalities would be tested alongside Step 2 of decompression

b. **Step 2:** Similar to above, we would start with smaller-scale tests (in the form of small, maybe even 1-element, UArray2's containing info2x2 structs) to make sure our functionality works correctly, before implemented larger-scale tests (larger UArray2's) to make sure it can handle large inputs. We would create a test file that creates a UArray2 of info2x2 structs and turns the information in those structs into 32-bit words, using field-update functions. We would then use the field-extraction functions to extract the elements of the word and return their values, checking them within the test file against our own expected outputs to ensure correctness. As an added test, we can also add plants into the info2x2 structs (in the form of values that are too large to be represented in their allotted bitspace) to test our width-test functions. These functionalities would be tested alongside step 1 of decompression

3. **Decompression**
    a. **Step 1:** We would create a test file that creates various differently-sized UArray2's (including 1-element UArray2's), with each element containing a 32-bit word. We would then run our field-extraction functions to get the values contained in the words; for each word, we would make sure that the values the functions returned matched our manually-calculated expected outputs by printing the returned values to standard output. These functionalities would be tested alongside step 2 of compression.
    b. **Step 2:** We would create a test file that creates various differently-sized, empty PPMs (including a 2-pixel-by-2-pixel file) and populates the array by decompressing a UArray2 of info2x2 structs. We would run through the elements of the array and printf the rgb values. Then calculate the correct rgb values manually (for smaller PPM files) and ensure that they match the output. For larger PPM files, we would test that the functions can handle larger files. These functionalities would be tested alongside Step 1 of compression. We would also utilize the "display" command on Terminal to make sure the image prints correctly.
4. **ppmdiff**
    a. Run both compression and decompression on several different-sized images and then use ppmdiff (developed in lab) in order to get the difference between the original image and the compressed->decompressed image. This would show us how good our compressor/decompressor is at preserving the image.