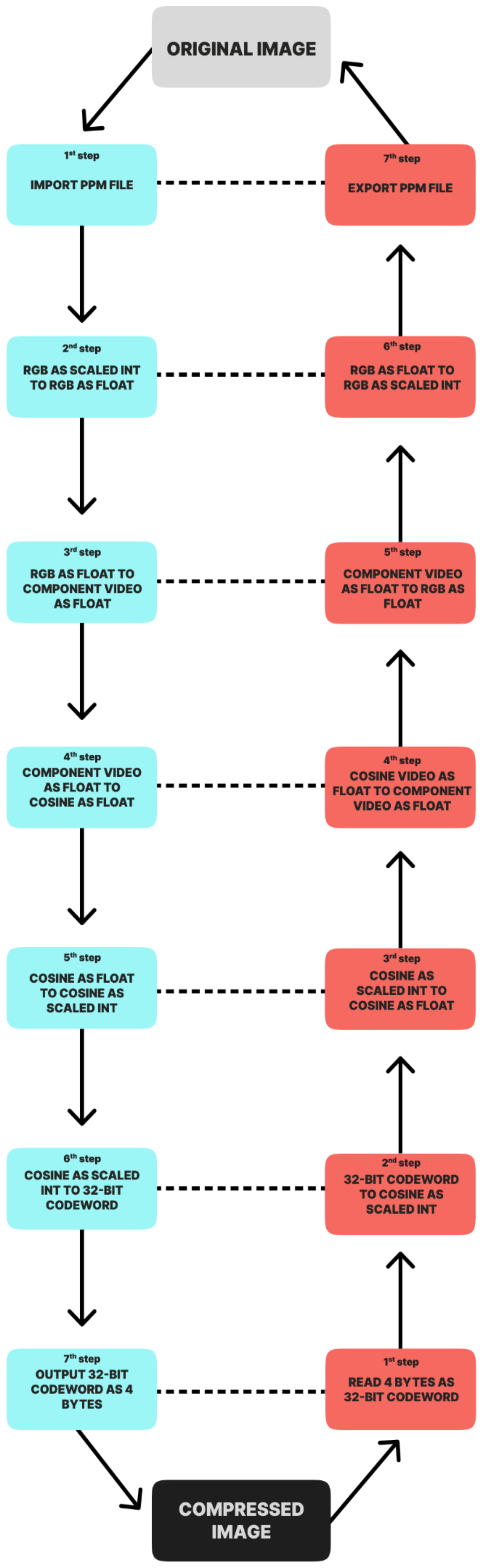
ORIGINAL IMAGE
1st step
IMPORT PPM FILE
2nd step
RGB AS SCALED INT TO RGB AS FLOAT
3rd step
RGB AS FLOAT TO COMPONENT VIDEO AS FLOAT
4th step
COMPONENT VIDEO AS FLOAT TO COSINE AS FLOAT
5th step
COSINE AS FLOAT TO COSINE AS SCALED INT
6th step
COSINE AS SCALED INT TO 32-BIT CODEWORD
7th step
OUTPUT 32-BIT CODEWORD AS 4 BYTES
COMPRESSED IMAGE
1st step
READ 4 BYTES AS 32-BIT CODEWORD
2nd step
32-BIT CODEWORD TO COSINE AS SCALED INT
3rd step
COSINE AS SCALED INT TO COSINE AS FLOAT
4th step
COSINE VIDEO AS FLOAT TO COMPONENT VIDEO AS FLOAT
5th step
COMPONENT VIDEO AS FLOAT TO RGB AS FLOAT
6th step
RGB AS FLOAT TO RGB AS SCALED INT
7th step
EXPORT PPM FILE

COMPRESSION STEPS

→ **STEP 1: Import PPM file**

Description: read PPM image from command line or standard input, trimming the image in case of odd height and/or width

Input: PPM image file from command line

Output: PPM image object populated with the opened file with RGB color space as scaled integers

Information status: in case the original image is trimmed to ensure an even parity, one row and/or column of data will be lost

→ **STEP 2: RGB color space as scaled integers to RGB color space as float**

Description: change the RGB color space int representation of the PPM image to a floating-point representation in RGB by using denominator of the image on every R-G-B of the pixels

Input: PPM image struct with width, height, denominator, and raster (2D array)

Output: 2D array holding a pixel struct consisting of floats

Information status: No data loss. For every image, all the integers will be divided by a constant element (the denominator), thus the information is still kept after the transformation

→ **STEP 3: RGB color space as float to component video color space (Y/Pb/Pr) as float**

Description: transform each pixel from RGB color space into a component video color space comprising the luminace (Y) and two side channels (Pb and Pr) through the provided linear formula

Input: PPM image with pixels in RGB-float format

Output: PPM image with pixels in (Y/Pb/Pr)-float format

Information status: No data loss. The conversion occurs under a linear transformation, thus the information is still reversible to its old format

→ **STEP 4: Component video color space (Y/Pb/Pr) as float to Discrete Cosine Transform space (Pb, Pr, a, b, c, d) as float**

Description: at every 4 pixels (2×2 blocks), the values are transformed to a new float cosine representation w/ given formula

Input: PPM image with pixels in (Y/Pb/Pr) component video space format

Output: PPM image in discrete cosine format (DCT) with the Pb and Pr average of the 4 pixels and the luminance (Y) of each pixel (a, b, c, d)

Information status: No data loss. DCT format can still be resversed to its old format given the linear transformations and for being floats

→ **STEP 5: DCT space (Pb, Pr, a, b, c, d) as float to DCT space (Pb, Pr, a, b, c, d) as scaled integers**

Description: transform each block of float pixels representation to a scaled integer representation

Input: PPM image in discrete cosine format with the Pb and Pr average of the 4 pixels and the luminance (Y) of each pixel (a, b, c, d) as floats

Output: PPM image with the same values now represented as scaled integers

Information status: Data loss. Converting float elements to integers may lead to the loss of decimal information

→ **STEP 6: DCT space as scaled integer to 32-bit codeword**

Description: transform each block of scaled integer pixels representation to a packed 32-bit codeword

Input: PPM image with the Pb and Pr average of the 4 pixels and luminance (Y) of each pixel (a, b, c, d) as scaled integers

Output: PPM image with 2D Array populated with codewords of every 2×2-block of scaled integers representation of the raster

Information status: the quantized values promoted data loss in the previous step. On step 6, the lossy conversion promoted on step 5 is just stored in a new format. No data loss

→ **STEP 7: Output 32-bit-codewords 2D Array as 4 bytes chars on disk big-endian order**

Description: print each block of scaled integer pixels as char representations in binary from the 32-bit codeword

Input: PPM image with 2D Array populated with codewords of every 2×2-block of scaled integers representation of the raster

Output: every codeword on the 2D Array printed as 4 chars in binary code on big-endian order on stdout

Information status: No data loss. 4-char format still keep the information on it

DECOMPRESSION STEPS

→ **STEP 1: Read 4 bytes (4 chars) to 32-bit-codeword 2D Array**

Description: read four chars (32 bits per time) to build a 2D Array of 32-bit codewords

Input: compressed image in chars as binary data

Output: PPM image with 2D Array populated with codewords of every 2×2-block of scaled integers representation of the raster

Information status: No data loss. 4-char format still keep the information on it

→ **STEP 2: 32-bit codeword to DCT space as scaled integer**

Description: transform each codeword into unpacked scaled integer representations of the pixels in DCT space

Input: PPM image with 2D Array populated with codewords of every 2×2-block of scaled integers representation of the raster

Output: PPM image with the Pb and Pr average of the 4 pixels and luminance (Y) of each pixel (a, b, c, d) as scaled integers

Information status: No data loss. The data here is solely unpacked and converted into a new format, no information is lost

→ **STEP 3: DCT space (Pb, Pr, a, b, c, d) as scaled integers to DCT space (Pb, Pr, a, b, c, d) as float**

Description: transform each block of scaled integer pixels representation to a float representation

Input: PPM image with values represented as scaled integers

Output: PPM image in discrete cosine format with the Pb and Pr average of the 4 pixels and the luminance (Y) of each pixel (a, b, c, d) as floats

Information status: No data loss. The conversion will keep the integrity of the former integer. However, the analogous step on Compression has compromised the original data

→ **STEP 4: DCT space (Pb, Pr, a, b, c, d) as float to Component video color space (Y/Pb/Pr) as float**

Description: at every 4 pixels (2×2 blocks), the values are transformed to a component video color space as floats

Input: PPM image in discrete cosine format (DCT) with the Pb and Pr average of the 4 pixels and the luminance (Y) of each pixel (a, b, c, d)

Output: PPM image with pixels in (Y/Pb/Pr) component video space format

Information status: No data loss. The linear transformations keep constant coefficients, so the float information is reversible

→ **STEP 5: Component video color space (Y/Pb/Pr) as float to RGB color space as float**

Description: transform each pixel from component video color space (Y/Pb/Pr) into a RGB color space

Input: PPM image with pixels in (Y/Pb/Pr)-float format

Output: PPM image with pixels in RGB-float format

Information status: No data loss. The conversion occurs under a linear transformation, thus the information is still reversible to its old format. Floats to floats also have no issue with preserving data.

→ **STEP 6: RGB color space as floats to RGB color space as scaled integers**

Description: change the RGB color space int representation of the PPM image to a scaled-integer representation in RGB

Input: PPM image struct with width, height, denominator, and raster (2D array) as floats

Output: 2D array holding a pixel struct consisting of integers

Information status: Data loss. For every image, float elements will be turned into integers, which may generate incongruences

→ **STEP 7: Export PPM file**

Description: write to stdout a decompressed PPM image with the retrieved RGB color space format as scaled integers

Input: PPM image object populated with the opened file with RGB color space as scaled integers

Output: printing of a decompressed PPM image

Information status: No data loss. Its analagous compression step may have lost a small portion of data.

Homework 4 - Design Document: Joao Quintanilha and Oliver Goldring

Implementation Plan

We built our architecture plan and strategy for testing through a “stepwise refinement” approach, as we have modularized all the steps for implementing an image compressor/decompressor. Based on this approach, as a compression step is written, its analogous decompression step is also written so that the image data gauging is constant. Our image will be stored in a PPM object, using a row-major UArray2_T data structure to hold the pixmap raster of the PPM image. Our main code will be performed on the file 40image.c, which will be responsible for parsing the command lines to decide whether it will be handling a Compression or a Decompression.

Following is the program implementation hierarchy with a review of tests and edge cases. More details about inputs, outputs, module description, and data loss considerations can be found in the overall design on the first page or in this Figma link.

1. **Implementation Step**

Compression Step 1: Import PPM File

- For this initial step, the best way of testing will be by providing different types of images and files to be accessed and handled by the function
- PPM file images of different sizes should be imported to check on the performance of the module
- PPM file images with even width/height and odd width/height, printing their width and height to check if the function is appropriately trimming the images with odd measures
- Run Valgrind to check if all the memory leak created by the module is reachable → check with different PPM images and also with the edge cases
- Use the next step to run more tests on the compression
- **edge cases:**
    - incorrect file (assert warning for not having successfully opened the provided file)
    - Incorrect PPM file (if the file is opened but the image is corrupted, or if it is not a PNM image, “pnm.h” resources will handle the errors)
    - PPM image with 1 pixel

2. **Implementation Step**

Decompression Step 7: Export PPM File

- This step will be written to test the first Compression Step, printing the PPM object to standard output after the file is open and read
- PPM diff can be run comparing the original image and the opened one, extracted on the first Compression Step 1. A small memory loss may be identified, considering that images with odd height/width are trimmed
- This is a very important step, as it will be responsible for providing outputs to check the correctness of the image through the cycle of compression and decompression
- Edge cases from Compression Step 1 will be also tested her
- Run Valgrind to check for memory leaks, considering that the usage of Hanson’s Data Structures needs to be freed

3. **Implementation Step**

Compression Step 2: RGB as scaled int to RGB as a float

- Once the color space elements are converted to floats, check if their width and height remain the same from the previous step
- Run Valgrind to make sure the old 2D array from the RGB as integers is properly freed
- An assertion must be included to ensure the provided PPM struct image is not a null representation
- Use the next step to run more tests on the compression

4. **Implementation Step**

Decompression Step 6: RGB as a float to RGB as an int

- It will be used to test step 3. It can be used to walk in the decompression pathway until the original image
- Decompress different sizes of images (small, medium, large) to check if they are working regardless of the size of the file
- PPM diff will provide details on the difference between the original image and its decompressed version. A slight difference may be found, as the trimming and the float-to-int conversion may generate incongruences
- An assertion must be included to ensure any data structure used is not a null representation
- Run Valgrind to ensure that any data structure used is properly freed

5. **Implementation Step**

Homework 4 - Design Document: Joao Quintanilha and Oliver Goldring

<u>Compression</u> **Step 3: RGB as a float to component video as a float**

- Print out data from the original and the compressed image, checking if the width and height of the transformed pixel map are appropriate. Now pixels are treated as 2x2 blocks of pixels, therefore this information must be considered while making the new arrays
- Run Valgrind to make sure the old 2D array is properly freed
- Provide an assertion to check if the input 2D array is not null
- Use the next step to run more tests on the compression

6. Implementation Step

<u>Decompression</u> **Step 5: Component video as a float to RGB as a float**

- Assertion to check the validity of the input 2D Array
- Decompress a compressed image to check how the details are behaving and also to see if any change is noticeable
- Width, height, and highest denominator on the raster
- PPM diff by decompressing different sizes of images, checking if the process is working and if any decompression can already be noticed → some additional data loss can be found here compared to the previous data loss, as the float-to-int transition can ignore less relevant decimal numbers
- Valgrind to ensure all data structures are being properly handled
- **edge cases:**
    - super saturated PPM image, with high numbers
    - PPM image with RGBs all in zero
    - PPM image with 2x2 pixels

7. Implementation Step

<u>Compression</u> **Step 4: Component video as a float to Cosine as a float**

- Print out data from the original and the compressed image, checking if the width and height of the transformed image raster are kept the same, considering that the data is still being treated in groups of 4 pixels per 2-D array element
- Assertion to check the validity of the input 2D array
- Valgrind to make sure the old 2D array is properly freed
- Use the next step to run more tests on the compression

8. Implementation Step

<u>Decompression</u> **Step 4: Cosine as a float to Component video as a float**

- Assertion to check the validity of the input 2D Array
- Valgrind to ensure the proper freeing of all data structures used
- Decompress a compressed image to check how the details are behaving
- PPM diff comparing original and decompressed image. The further in the pathway we go, the more we expect to see data loss
- All the edge cases aforementioned can be also tested with this new step

9. Implementation Step

<u>Compression</u> **Step 5: Cosine as a float to Cosine as an int**

- Print out data from the original and the compressed image, checking the width and height of the transformed raster. It is important to keep in mind the measures in the previous compression.
- Assertion to check the validity of the 2D array
- Valgrind to make sure the old 2D array is properly freed
- Use the next step to run more tests on the compression

10. Implementation Step

<u>Decompression</u> **Step 3: Cosine as an int to Cosine as a float**

- Decompress images from the analogous compression step, checking specific details from the decompressed image
- Since data loss is expected with quantization on its analogous compression part, PPM diff will indicate after the decompression a higher difference between the images
- Assertion to ensure the use of valid 2D arrays
- Valgrind to ensure the proper freeing of the used data structures
- Check if the int-float conversion is working properly (we can choose a few blocks and look specifically to the pixels being held there

11. Implementation Step

<u>Compression</u> **Step 6: Cosine as an int to 32-bit codeword**

- Print out data from both images to see if any relevant change is seen. A few pixels from both images should also be printed to compare and assure the conversion into codewords is successful
- An assertion to ensure the 2D array input is valid

- Valgrind to ensure the freeing of all data structures
- Use the next step to run more tests on the compression

12. Implementation Step

<u>Decompression</u> Step 2: 32-bit codeword to Cosine as an int

- Decompress images from the analogous compression step, checking specific details when printing back the image after all the compression performed so far. It is valid to check a few specific pixels on the image after the migration from one format to the other, checking if the reverse formula is being properly applied
- PPM diff to evaluate if the data information is being kept
- Assertion to ensure the use of valid 2D arrays
- Valgrind to ensure the freeing of the data structures

13. Implementation Step

<u>Compression</u> Step 7: Output 32-bit codeword as 4-bytes of chars

- Once it reaches the final step, the image is totally compressed. We will print the compression data to standard output and check if some of the 4-char elements match to its 32-bit codeword version. Additionally, width and height will also be checked to make sure the size is appropriate
- Assertion to ensure a valid 2D array input
- Valgrind to ensure the freeing of all data structures
- Use the next step to run more tests on the compression

14. Implementation Step

<u>Decompression</u> Step 1: Read 4-bytes of chars as 32-bit codewords

- Assertion to ensure valid 2D array input
- Valgrind to ensure the freeing of all data structures
- This step will be used to decode the compressed image and pass it throught the decompression pathway, until it gets to its “decompressed” version. PPM diff will be used to compare the final product image to the original one, making sure that some compression was performed