Design Doc: Alejandra Sabater (asabat01), Darius-Stefan Iavorschi (diavor01)

1. **Implementation plan for Part C, ppmtrans:**

Arguments provided to ppmtrans.c on the command line or through standard input:

- rotate {0, 90, 180, 270 degrees, flip horizontal or vertical, transpose}: determines the angle at which we want to rotate the matrix or flip the matrix vertically, horizontally, or diagonally (transpose)
    - If nothing is provided on the command line or standard input, defaults to a 0 degrees rotation
- {row, col, block} major: determines the iteration used to go through the image
    - If nothing is provided, the program defaults to map_dafult, which uses a default order that has good locality
- Time time_file: Create timing data and store the data in the file named (defaults to none)
- Filename: the input file name (defaults to none)

Ppmtrans.c already has for loop and if statements to identify if we should be using a UArray2b or UArray2 and creates a map pointer that we can use.

Set up the array of pixels for the picture: *NOTE: the blue text is how we are going to test each step.*

Once the for loop, where command line arguments are processed, is exited:

1. If nothing is given in the command line, read from STDIN
2. If something is given on the command line, check to see if the last argument is the file name
    a. If the last argument in the command line is not a filename, then the command line was wrong and we raise an error.
    b. If the last element is a file, continue to the next step.
3. Open the file, check if file was correctly opened
    Test by giving the program a file that does not exist
    Test by giving the program an empty file
4. Using pnm.h functions to read in the file:
    a. Pnm_ppm Pnm_ppmread(FILE *fp, A2Methods_T methods);
        i. This function reads a file using the given methods and returns a pixmap containing a 2D array of the type returned by 'methods->new'
        ii. This is our array of pixels that we will them transform according to what was specified on the command line

Give the program multiple different test files. Print out the width, height, size, and blocksize of the array and check that it is what we expect

5. Now that we have a 2D array of the picture pixels, we can transform the image using the transformations defined below
6. To perform these transformations, we have to create another 2D array, where we can translate each pixel and put it into a new place in a new 2D array that is the same (UArray2 or UArray2b) as the original, which represents the newly transformed picture.
   Check to make sure that new 2D array has been made correctly
7. We will iterate through our untransformed picture, grab one pixel at a time, and place it in its new location on the newly transformed picture array
   To test that we are correctly grabbing each pixel, we will give the program a small test file and we will print it out as the code runs. This will help confirm that the pixels in the array are correctly placed by the Pnm_ppmread and that we correctly traverse through in the order specified
8. The traversing to grab each pixel is done according to what was specified on the command line or through STDIN:

Traversing:

- Row major: traverses elements in the first row up, then it moves to the second row and traverses through those elements, and so on.
  - At each element, we apply the apply function, which will perform one of the transformations.
    To make sure this works, give the program a file and make the apply function so that it prints out each element that is visited. This way we can compare what the expected traversing should look like and we can see if the applied function is correctly being applied to each element.

- Column major: traverses elements in the first column, then moves to the second column and traverses through those elements, and so on.
  - At each element, we apply the apply function, which will perform one of the transformations.
    To make sure this works, give the program a file and make the apply function so that it prints out each element that is visited. This way we can compare what the expected traversing should look like and we can see if the applied function is correctly being applied to each element.

- Block major: traverses through block by block, and visits all cells in one block before moving on to the next block.

- At each element, we apply the apply function, which will perform one of the transformations

    To make sure this works, give the program a file and make the apply function so that it prints out each element that is visited. This way we can compare what the expected traversing should look like and we can see if the applied function is correctly being applied to each element.

9. Below we identify where on the new 2D array each of the pixels will be placed, with what equation:

Possible Picture Transformations:

- Flip horizontally: Mirrors the image horizontally (left-right)
    - The image is **reflected across the vertical axis**
    - Mapping formula

        newMatrix[row][width - col - 1] = oldMatrix[row][col].

        To ensure that this correctly places the pixel in the new array, we can print out the array index that we are placing in to make sure it is correct. Then we can use the AT method to retrieve the element at that index to check that the element is correctly placed.

- Flip vertically: Mirrors the image vertically (top-bottom)
    - The image is **reflected across the horizontal axis**
    - Mapping formula

        newMatrix[height - row - 1][col] = oldMatrix[row][col].

        To make sure that this correctly places the pixel in the new array, we can print out the array index that we are placing in to make sure it is correct. Then we can use the AT method to retrieve the element at that index to check that the element is correctly placed.

- Transpose: Transposes the image (across UL-to-LR axis)
    - The image is **transposed**, swapping rows and columns
    - Mapping formula:

        newMatrix[col][row] = oldMatrix[row][col].

        To make sure that the image is correctly transposed, print out the index on the new picture array. Also use the AT method to retrieve the element at that index to check that the element is correctly placed.

- Rotate 90 (clockwise):
    - The image is first **transposed** (swap rows and columns)
    - The result is then **flipped horizontally**
    - Then the mapping formula is:

        newMatrix[col][height - row - 1] = oldMatrix[row][col].

      To make sure that the image is correctly rotated 90 degrees, print out the index on the new picture array. Also, use the AT method to retrieve the element at that index to check that the element is correctly placed.

- Rotate 180:
    - The image is flipped **both horizontally and vertically**
    - Mapping formula:

        newMatrix[height - row - 1][width - col - 1] = oldMatrix[row][col]

      To make sure that the image is correctly rotated, print out the index on the new picture array. Also, use the AT method to retrieve the element at that index to check that the element is correctly placed.

- Rotate 270 (clockwise): *(or 90 counter-clockwise)*
    - The image is first **transposed**
    - The result is then **flipped vertically**
    - Mapping formula:

        newMatrix[width - col - 1][row] = oldMatrix[row][col]

      To make sure that the image is correctly rotated, print out the index on the new picture array. Also, use the AT method to retrieve the element at that index to check that the element is correctly placed.
- Rotate 0:
    - Leave the image unchanged

      Print out the original image and then the new image and compare. Make sure they are identical.

- Time <timing_file>: Create timing data (see Section 1.5 below) and store the data in the file named <timing_file>.

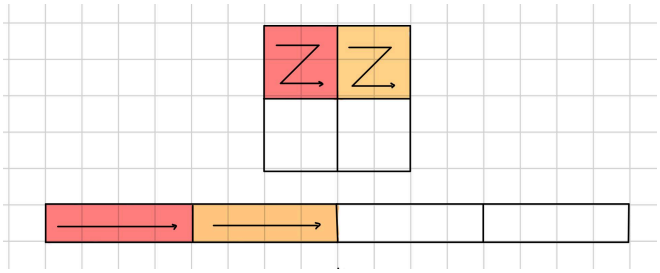
10. Once every pixel has been grabbed from the original 2d array and has found its new location in the new 2d array, we will free the memory used up by the original array.

11. Then we will iterate through the transformed picture and write the transformed image to standard output, in binary ppm format.
    Check and compare with the original image to make sure that the correct translations have been applied.
12. Finally, we will free the memory that the new picture array was taking up and return success.
    To test that there are no memory leaks, run with Valgrind

2. Estimates and explanations for Part D:

Note: We are defining the UArray2 matrix as a 1-dimensional UArray in which the index is calculated using the formula:
index = row * width + col, which favors row-major access.

We are defining the UArray2b matrix as a 1-dimensional UArray where blocks are stored contiguously in a long array. The index is calculated using the formula
index = row * width * blocksize * blocksize + col * blocksize * blocksize + index_block, (where row and col refer to the position of a block, not a pixel)

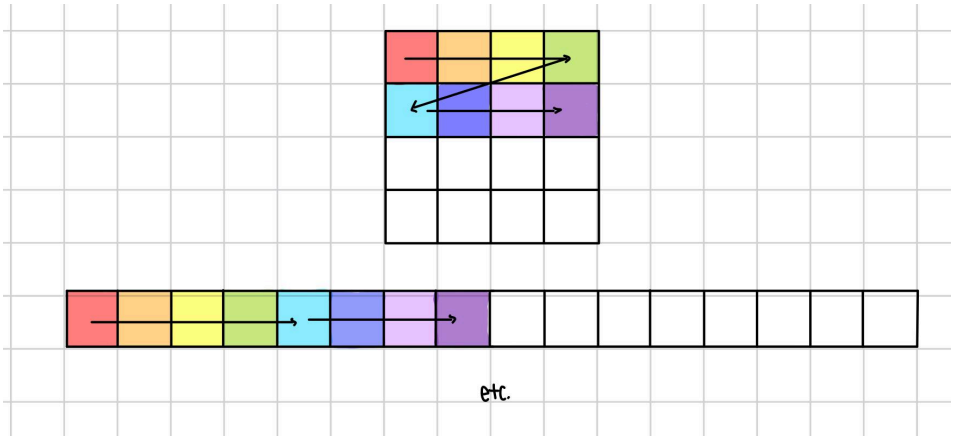


180 and 90-degree rotation Blocked access (UArray2b): **#1**

- All elements are stored contiguously in blocks that are stored near each other in memory, resulting in very good locality
- We are retrieving pixels from one block and storing them in another block before moving on to other pixels.

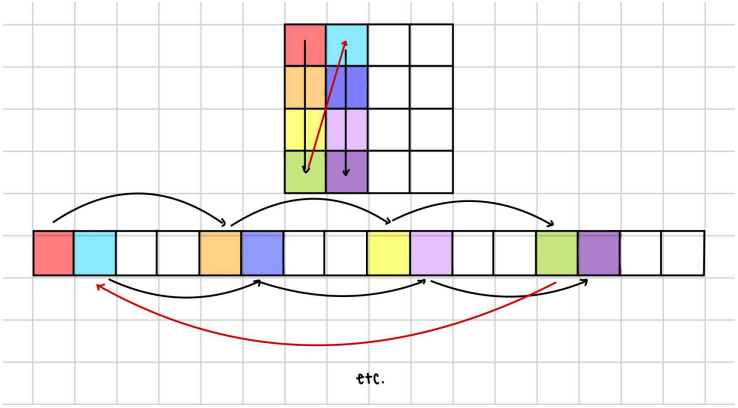
180 and 90-degree rotation Row-major access (UArray2): **#1**

- Elements are stored contiguously in each row. Using row access would guarantee we are reading all the elements from one cache line before accessing the next one.



180 and 90-degree rotation Column-major access (UArray2): **#2**

- Elements are not stored contiguous in one column. The program would have to jump around the array, pulling a new cache line every time a new pixel is read. We expect the cache miss rate to be very high.



Note: We assume that rotation is irrelevant to the performance of read operations. The main impact on performance is determined by the traversal method.

If we are only considering the reading aspect of our six operations, we estimate that our 180 and 90-degree Blocked Access and our 180 and 90-degree row access will perform approximately the same and have approximately the same cache misses (the block major encounters a small memory wastage. The difference becomes negligible for large arrays). This is because each block is stored continuously in the UArray2b, and when we read in each pixel, there is no jumping around the array.

Our drawing above shows the way the pixels and blocks will be read. In row-major access all pixels are stored in order from left to right in the array already, so reading in this order is efficient and also involves no jumps.

Finally, we gave 180 and 90-degree column major access last place. When we read in column-major order, the program has to jump from one place to another in the array, causing

more cache misses, as the elements in different lines are not pulled into the cache automatically as a pixel is read. We expect the cache hit rate to be low.