Design: Locality

**PPMTrans Implementation**

- Interactions
  We expect to implement polymorphic rotations in our PPMTrans program. We intend to do this through the method suite of A2Methods which is declared in both versions of the arrays we are utilizing in this program. The method suite will have access to all functions each of the arrays through function pointers. However if a client attempts to use a function that is not at all applicable to an array (i.e. row or col major on a UArray2B array), said function is set to null and will raise an exception. We also intend to keep all of our rotation functions in our PPMTrans program–though this comes with less abstraction and modularity we find that it will be easier to test and follow code on our end.
- Functions

- PPM Reader
  - We intend to use the Hanson interface to read in the file to a PPM file. We will catch any asserts thrown by said interface as well as run through some test cases we have outlined below.
  - **TESTING**: We will pass in arbitrary files ranging from simple 3x3 images to the massive files that we are given - some of which from the bitonal directory. We will also use our own images that have been recently cropped to see what will happen - we assume that our code will be able to still obtain the correct dimensions even though they have just been edited. We will also pass in corrupted images, standard input, and an empty image file. We must also catch the asserts from hanson pnm reader in case there is an issue reading the file (either header or not matching dimensions) so that our program does not crash out early and not free our heap. Finally we will run valgrind every 5-10 lines of code to make sure that everywhere we allocate we also free somewhere.
  - After we have read in the file, based on the starter code given the default array is the UArray2 so we will be using that to hold the image unless block-major order is called in which case we will call the block-major order function and create a UArray2B from reading in from a UArray2.
    - We intend to use a similar solution to that of 90 outlined below where we use the at and mapping function along with the math provided in the spec to create said blocked array. In this case we will also use the new function and pass in the sizeof(element) for both block and size.
- Rotate 90
  - UArray2_new(int width, int height, int size)
    We intend to create a new UArray to copy over elements into their new indices. In doing this we will have access to the untampered with element addresses. We should not need to test this function too thoroughly because we will be using the answers from iii that we are given.
  - UArray_map_() (if no arguments are passed we will default to row_major)
    We will use the mapping functions to find the elements at a specific index and call the apply function for said indices. In our apply function we will call the UArray2_at() function on the new array given the 90* tweaked indices and store the return in a void pointer and cast it to an integer which we will set to the value of the current element. For testing our

mapping functions we will check the dimensions are correct by asserting that the dimensions we put in are returned to us by our getter functions in a testing file. We also intend to assert that the sizes of the blocked array are implemented correctly as well as the null case, when Hanson throws an exception. We will run valgrind every 10-15 lines of code and especially after allocating an element on the heap. We will keep a list of these elements. We also intend to print a counter for every index in the array and ensure that we are mapping in the correct major order.

- applyNine(UArray2d rotateNine, int col, int row, void *elem, void *cl)
    - UArray rotateNine is the new array that will represent the elements for the rotated image
    - Col is the original column corresponding to the element location
        - We will be calculating the rotated column coordinate in the applyNine function via the math given in the spec where i is equal to the column [(h-j-1)]
    - Row is the original row corresponding to the element location
        - We will be calculating the rotated row coordinate in the applyNine function via the math given in the spec where j is equal to the row [i]
    - Elem is the void pointer pointing to the element in the original array
        - We intend to create a temporary pointer which holds the address of the rotated index in the new array and cast the pointer to an integer and have the temporary void pointer hold the value at the original index
    - **Testing our apply function**: Since our apply function to rotate images is the last function we write and we have incrementally tested all of our previous code we can test by simply giving our program edge case rotations to preform and if something is working improperly we know there is an error in our apply function. To test our rotation worked correctly on the smaller images we will hard code a flipped version of the image and diff test it with the output from our program.
- UArray_at (int mathi, int mathj, UArray2d rotateNine)
    - We intend to call the at function in our apply with the calculated row and col integers and the new array for the rotation to find the address of the new indice and return the address to the apply function so we can set a value to that address

- Rotate 180
    - We will use the same exact structure as our Rotate 90 function: building a new UArray2 and using a mapping function to visit every index of our original UArray2 or UArray2b and insert the value at that location into the correct spot in our new array but using different math so that the image will rotate 180 [pixel (i, j) becomes pixel (w – i – 1, h – j – 1].
- Rotate 270
    - Going to use the same method as 90 and 180 but in this case, rotation comes about from rotating the image 90 degrees and then 180.

- Rotate 0
    - Will just return the image
- Flip horizontal
    - Find the negative number (0, 0 becomes height of array, height of array). We are still workshopping this math.
- Flip vertical
    - Find the negative number (0, 0 becomes width of array, width of array). We are still workshopping this math.
- Transpose
    - We don't have the math for this.
- Time <timing_file>
    - We intend to use the provided code to check how long each rotation takes. We will use our assumptions at the bottom of this code to verify that things are implemented efficiently and/or to double-check our assumptions (with TA help and references to the book).

**Testing**

Our test cases have been outlined in the implementation portion of this document.

**Cache Hit Rate**

| | **Row-major access (UArray2)** | **Column-major access (UArray2)** | **Blocked access (UArray2b)** |
|---|---|---|---|
| **90-degree rotation** | **3** | **4** | **1** |
| **180-degree rotation** | **4** | **2** | **1** |

Blocked Access

We believe that blocked access has the highest hit rate because of the elements being stored contiguous in memory. The only instance in which we believe that cache efficiency is affected would concern block size taking up far more space than it has to. In our program to ensure optimal space we will be calculating block size from the size of each element in our array.

- 90-degree rotation:

  We believe that because blocked is the best cache hit rate wise the same will follow for the 90 degree rotation. In this case there is better spatial locality due to the way iteration is handled in block-major order (i.e. contiguous memory access). In our code, we intend to create a new array and copy back and forth between the original. This process is helped with the blocked major access because of the aforementioned explanation.

- 180-degree rotation:

  We believe that there is not really a difference between the 90 and 180 in this case. However in this assignment because the math to rotate the image 180 degrees involves accessing an additional variable which may decrease the cache hit rate (very slightly).

Column-major access

We believe that column major access is the next best in terms of cache hit rate because the spatial locality is analogous with how the array is stored (c, r).

- 90-degree rotation
  We believe that the 90-degree rotation is not as good (cache-hit-rate-wise) as blocked or col-180 because though the columns are contiguous in memory, the writing into the rows in reverse order requires jumping around in memory which does not align with the original image layout
- 180-degree rotation
  We believe that the 180-degree rotation in the col-major access case is better than the 90-degree because it does not require jumping around in memory.

Row-major access

We believe that row major has the worst cache hit rate because the spatial locality is not at all analogous to how the array is stored in memory (c, r).

- 90-degree rotation
  We believe that the 90-degree rotation is better than the 90-degree column rotation. This operation aligns well with the original row-major layout, resulting in what we believe as better spatial locality and a higher cache hit rate compared to column-major access for the same operation. However, it still involves traversing rows in reverse order, which does still impact cache performance, from our understanding.
- 180-degree rotation
  Similar to the 90-degree rotation, a 180-degree rotation using row-major access involves reversing both rows and columns while creating a new matrix. The spatial locality may be better compared to column-major access, especially if the original data has strong row-wise patterns. But because we are considering this in a general sense we have decided to consider it the worst way rotation in this bunch.

**UArray2B Implementation (for our own use)**

We intend to build out the 2D array by stacking 2 arrays. We intend to create one large array corresponding to the blocks and then a smaller array for accessing the image elements. To create the larger array we will use a math function to find the width and height: ceiling(width/blocksize) for the width and then ceiling(height/blocksize) for the height. We intend to iterate through the array using col x row for our (x,y). For the 64K block we will initialise an 11x11 array for the larger array (as the number of bytes is 1024 and our main array is comprised of pointers).