Design Document

HW #3: Locality

By Jared Lieberman and Jordan Stone

Part A:

1. ***What is the abstract thing you are trying to represent?*** **Often the answer will be in terms of sets, sequences, fi- nite maps, and/or objects in the world. Always the answer is client-oriented.**

We are storing any type of data in a blocked 2Darray.

2. ***What functions will you offer, and what are the contracts those functions must meet?***

```
extern T    UArray2b_new (int width, int height, int size, int blocksize);
```

This function will create a blocked two dimensional array of type UArray2_T. Inside each element (block) of the array is a one dimensional array of type UArray_T. The function returns the object to the client. The function arguments are the dimensions of the two dimensional array in term of the number of columns and the number of rows desired by the client. Each cell inside the block in the two dimensional array will occupy `size` bytes of memory. The bytes in the array are initialized to zero. The `size` parameter must include any additional memory padding. The `blocksize` the function to create the length of the one dimensional array with a length of `blocksize * blocksize.`

```
extern T    UArray2b_new_64K_block(int width, int height, int size);
```

This function will perform similarly to UArray2b_new; it will create a two two dimensional array of type UArray2_T, each element of which consists of a one dimensional array of type UArray_T. Each block in the UArray_2T two dimensional array contains 64 bytes of memory. The width and height arguments determine the dimensions of this array. The size argument determines the number of bytes in each cell in a block. This function defaults the blocksize for the client. If a single cell does not fit in the 64KB block, the blocksize will be 1.

```
extern void  UArray2b_free     (T *array2b);
```

This function deallocates memory for both the two dimensional array and the one dimensional array and deletes the object. The client must pass in the pointer to the object.

```
extern int   UArray2b_width    (T  array2b);
```

This function takes the UArray2b_T array as an argument and returns the number of columns in the array.

```
extern int   UArray2b_height   (T  array2b);
```
This function takes the UArray2b_T array as an argument and returns the number of rows in the array.

```
extern int   UArray2b_size     (T  array2b);
```
This function returns the number of bytes required by a single cell in a block.

```
extern int   UArray2b_blocksize(T  array2b);
```
This function returns the size of one side of a block.

```
extern void *UArray2b_at(T array2b, int column, int row);
```
This function returns a pointer to a single cell inside the two dimensional array. The client can access the element by casting the pointer that the function returns and then dereferencing the pointer. This column and row arguments determine which cell to access.

```
      extern void  UArray2b_map(T array2b, void apply(int col, int row, T
array2b, void *elem, void *cl), void *cl);
```
The client calls this function by passing in the two dimensional array, an apply function that the client implements, and the closure that the client provides. This function maps over all of the cells in one block before moving to another block. While visiting each element of the array, it calls and apply function which is applied to the element.

3. ***What examples do you have of what the functions are supposed to do?***

```
UArray2b_T sample_arr = UArray2b_new(3, 2, 2, 4);
// this makes a 3x2 UArray2b_T array that holds blocks with blocksize of 4 //
and cell with a size of 2 bytes.
UArray2b_T sample_arr_64 = UArray2b_new_64K_block(3, 2, 32);
// this makes a 3x2 UArray2b_T array that holds blocks with each cell
// holding 32 bytes of memory.
Int w = UArray2_width(sample_arr);
// this sets a variable w to the width of the UArray2_T array
int h = UArray2_height(sample_arr);
// this sets a variable h to the height of the UArray2_T array
int size = UArray2_size(sample_arr);
// this sets a variable s to the size of an cell in each block
int b = UArray2b_blocksize(sample_array);
//this set a variable b to the blocksize of a block
void *element = UArray2_at(sample_arr, 1, 1);
char *c = 'a';
*((char *)element) = c;
```

```
// this returns a void pointer to element (1,1) in the UArray2b_T. And //then
changes the value that the void pointer points to in the array
UArray2b_map(sample_arr, apply, NULL);
// maps through array and calls apply on each cell, going through each
// cell in a block before moving to the next block
UArray2b_free(&sample_arr);
// this frees the UArray2b
```

4. ***What representation will you use, and what invariants will it satisfy?*** **(This question is especially important to answer *precisely*.)**

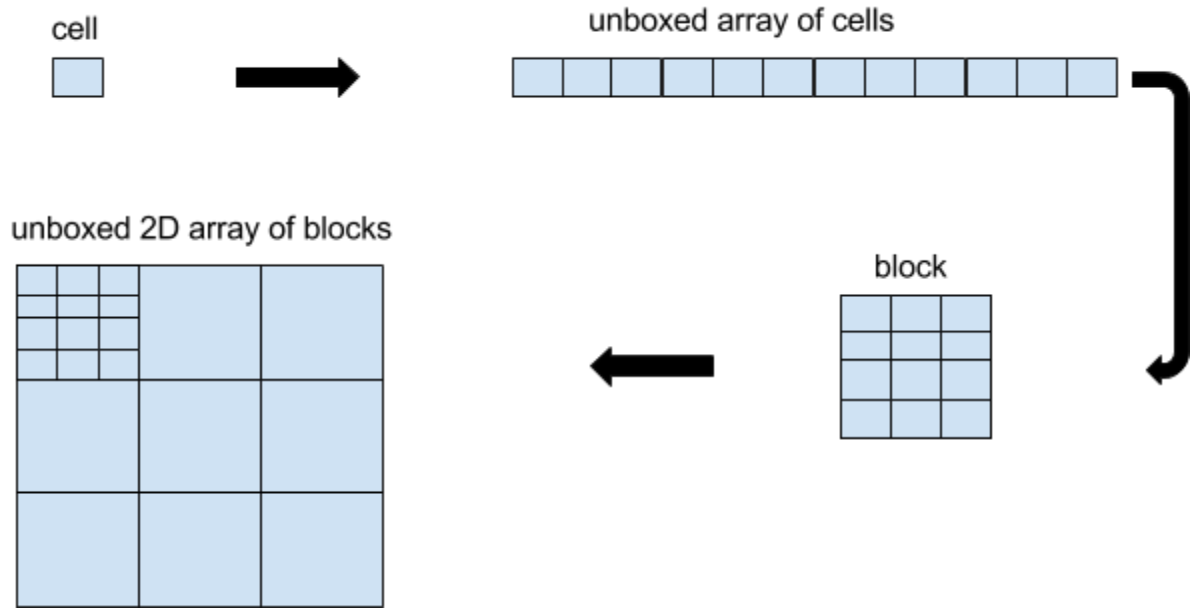
To represent our two dimensional UArray2_T array of blocks, we will use an unboxed two dimensional array with each element representing a block. Each block will be represented by a one dimension UArray_T. The Uarray2b_T representation will consist of these two data representations.

The invariants for this representation are as follows. The memory addresses of data stored inside a single block will be close to one another (contiguous memory). This ADT will always be able to hold any type of data type because it is polymorphic. The maximum number of cells inside a block is `blocksize * blocksize.` Another invariant is a specific set of operations to locate a specific cell in the UArray2b_T representation. Given a cell index (col, row), we find the specific block number at index (col / blocksize, row / blocksize). Then, within that block, we use this new index to find the specific cell in the block with (blocksize * (col % blocksize) + row % blocksize).

To translate a location within our representation back to pixel coordinates, you have the index of the location inside the UArray_T and you have the index of the block that holds the location inside the UArray2_T. The formula for translating the location to the master cell two dimensional index is as follows: (((location index) % blocksize) + (blocksize * block_col_index) , ((location index) % blocksize) + (blocksize * block_row_index)). If the expression is simplified, it will result in the desired cell index from the location.

5. ***How does an object in your representation correspond to an object in the world of ideas?* That is, what is the in- verse mapping from any particular representation to the abstraction that it represents. This question is also especially important to answer *precisely*.**

An object in our representation is a single cell inside an unboxed array. Each unboxed array represents a block. Lastly all the blocks are represented as a two dimensional unboxed array



6. ***What test cases have you devised?***

-We plan to test the array of blocks by calling the creation of the array as a single dimensional array of blocks.

-We will test the implementation by calling the creation of an array with various number of cells. These will include certain corner cases in which the number of cells in the UArray2b_T representation is less than the UArray2_T representation can hold (blocksize * blocksize * UArray2_width * UArray2_height). We will ensure the program stores and manipulates data correctly when this is the case.

-We will test the implementation by calling the creation of an array with cell size of 0.

-We will try to break the program by trying to test the UArray2b_at function by calling the function with incorrect dimensions.

-We will call `Uarray2b_new` and check the width and the height manually, and then with the width and height functions.

-We will test the mapping functions by writing our own appy function that prints out the elements in the order they are accessed to make sure that the map visits the elements in the correct order.

-We will call the creation of `UArray2b_new_64K_block` with various input sizes(including zero) and especially cases where the size exceeds 64KB.

7. ***What programming idioms will you need?***

We will use the idiom of using an abstraction defined in an interface. We will use the idiom for using unboxed arrays. We will use the “getting rid of unused variable warnings” idiom. We will use the idiom of storing values in an unboxed array. Furthermore, we will use the idioms for “abbreviating structure types” and “allocating memory”. We will use the idiom for `void **` pointers. Generally we use the idiom of separating the interface from the implementation to prevent the client from accessing the implementation.

Part C:

1. ***What problem are you trying to solve?***

We need to be able to execute simple image transformations.

2. ***What example inputs will help illuminate the problem?***

Useful inputs will be image files that are incorrectly oriented. They are will be rotated and transformed in different ways.

3. ***What example outputs go with those example inputs?***

The example outputs that will go with these example inputs will be the normal images, or correctly oriented versions of the sample images.

4. ***Into what steps or subproblems can you break down the problem?***

We can break down the problem into numerous sub problems including reading in the image file and performing the different transformations while optimizing locality. Lastly, we will need to output the transformed image.

5. ***What data are in each subproblem?***

Reading in the image: The file stream.

Different Transformations: The unboxed two dimensional arrays (both blocked and normal) and the image data stored in binary ppm format inside the arrays.

Output: The unboxed two dimensional arrays (both blocked and normal) and the image data stored in binary ppm format inside the arrays.

6. ***What code or algorithms go with that data?***

For the mapping functions, the algorithm is different for each transformation. For 90 and 270-degree transformations, the row indices map to different column indices and vice versa. For the 180-degree transformation, the row indices map to different row indices and the same for column indices. For the 0-degree rotation, the image is unchanged.

7. ***What abstractions will you use to help solve the problem?***

We will be using the A2Methods abstraction to store and manipulate data within the program. In addition, we will delegate different responsibilities of the program as a whole, divided by both function and purpose. Here we plan to make use of procedural abstraction to help minimize the number of lines of code to implement each of the transformations. We will break apart each transformation into sub components of manipulating the elements in the two dimensional arrays into single steps. Therefore, when a transformation is requested, the part of

the program responsible for completing that transformation will call all of the necessary functions that comprise a single deconstructed transformation.

8. ***If you have to create new abstractions, what are their design checklists?***

Not applicable.

9. ***What invariant properties should hold during the solution of the problem?***

The main invariant of the program is that `ppmtrans.c` will only interact with the A2Methods abstraction to manipulate the data to solve the problem. `ppmtrans.c` will not directly call either of the two abstract data types, UArray2_T or UArray2b_T. Additionally, another invariant is the geometric calculations needed to perform the image transformations. With an original image of size (w * h) in a 90-degree transformation pixel (i,j) becomes (h-j-1, i). Secondly, in a 180-degree transformation pixel (i,j) becomes (w-i-1, h-j-1).

10. ***What algorithms might help solve the problem?***

Much of the algorithm that lies within the problem of executing simple image transformations will include important functionality from the mapping functions. There will be different algorithms for each transformation, and because we will be writing the image to standard output, the mapping functions will need specific methods of accessing certain other elements when the function has reached that element. Specific transformation calculations are noted in the invariants section.

11. ***What are the major components of your program, and what are their interfaces?***
Components include functions as well as abstract data types. An interface includes contracts as well as function prototypes.

The components of our program include numerous functions that store and manipulate the data use the A2Methods suite. The interface for the mechanism to manipulate the data is `a2methods.h` . The program `ppmtrans.c` will not have its own interface as it is a client of `a2method` .

12. ***How do the components in your program interact? That is, what is the architecture of your program?***

The transformation functions of `ppmtrans.c` will interact with `a2methods.h` in order to perform the image transformations. Since `ppmtrans.c` is a client of `a2methods`, the specific individual data manipulation abstraction is hid from `ppmtrans.c` . The suite of a2methods will interact with two interfaces to manipulate both blocked and unblocked representations of data.

13. ***What test cases will you use to convince yourself that your program works?***

We will use cases where there is a clear difference in the input vs. the correct output.
We will use cases where the image file is very large.
We will use cases where the image file is very small(even one pixel small).

We will use cases where the image is symmetrical about the x and y axis and perform rotations.

14. ***What arguments will you use to convince a skeptical audience that your program works?***
We will show the skeptical audience that the program has a logical structure and performs the image transformations and we will show that the program passes numerous test cases that hone in on the corner cases. This is important in understanding that the program can handle many variations of data.