Data Structure Project Final Report

**Image Processing in C**

**Academic Adviser**

Prof. Saejoon Kim, Sogang Univ.

**Student Team #3**

Dohyun Kim(20141505), Sogang Univ.

Younjoon Chung(20141579), Sogang Univ.

Kwanghee Choi(20141589), Sogang Univ.

**- Contents -**

1. Problem Settings

2. Problem Analysis

3. Data Structure Implementation

4. Image Processing Implementation in Quadtree

5. Technology Analysis

6. Development Cycle Analysis

7. Conclusion

**1. Problem Settings**

BMP is the most frequently used image format when processing images, because bmp images store its rgb pixels in a two-dimensional array. It is followed that every image processing system uses two-dimensional array. But, as every data structures does, it has pros and cons.

We could observe that this data structure can be very inefficient in some specific situations, such as representing sparse images, as this data structure saves pixel data one by one, similar to the original matrix in the data structures class, so it was natural that we search for new data structures, like sparse matrix representation.

For example, let’s say that an image with a height and width of 1024 pixels are only made with one color; uniformly colored in black. This image, in fact, only need three informations: height, width, and color of the image. But traditional 2-dimensional fixed array stores approximately one million pixels in a 1024 x 1024 array, in which all of the pixels have the same value. This inefficiency goes same with image operations, such as filtering. If we use gaussian blur for uniformly colored image, the result of blurring is the same with original image, which attends unnecessary operations.

**2. Problem Analysis**

We found that Data structure named quadtrees is made to solve this inefficiency. This data structure is mainly used in representing sparse images, such as geometrical shapes or dots on the coordinate plane, or computing an intersection or union between two images.

Our project will expand the usage of the quadtrees to representing images by implementing basic operations such as operations using masks.

The reason for implementing this particular data structure for image operation is simple. The worst time complexity for reading one pixel is O(log2n), when the width and height of image is n. It can be disadvantageous in some situations with dense images, as 2-dimensional array representation requires O(1) for reading. But for sparse images, it is far more advantageous than array as this representation does not visit every pixel in the image, and hence skipping unnecessary calculations.

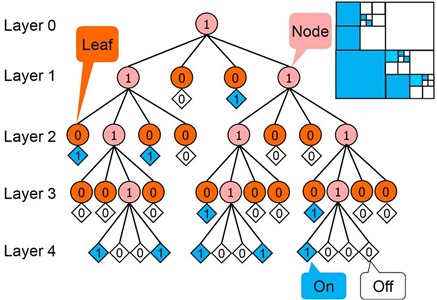


Image representation for data structure of quadtree

(Representing halftoned image)

Quadtree is designed to store minimal required information possible. The way to construct a quadtree from a singular image is the following. If the whole segment is of the same value, or it is one pixel, then make a leaf node and end the procedure. If not, divide it into four seperate nodes (2 x 2 squares) and repeat the procedure recursively.

Maximum number of layers is log2n, when the width and height of the image is n pixels. Therefore the time complexity for reading one node is naturally O(log2n). This is because the maximum number of nodes of the layer k is 4k. The worst case possible is when the image has the distribution similar to that of a checker board, and in this case, total number of nodes is n2 + n2/4 + n2/42 + … + 1; therefore, space complexity is O(4n/3).

**3. Data Structure Implementation**

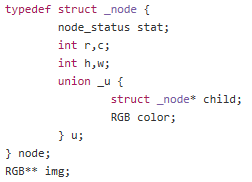
We implemented quadtree without using pointers expilcitly, namely pointing the data using pointers. We used pointers only to expand the node array. This style is similar with JAVA style objects, but without classes or behaviors. Tree node is not called by the pointer pointing to the node, but the node itself, so if the node is manipulated, node have to be copied by value. This is the key difference from the JAVA objects and our implementation.

By implementing by this style of data structure, it is a quite safe method, as memory leakage or pointing in the wrong memory does not occur, as this node moves as a whole data structure. And also this data structure is not that big, as child nodes are elements of the node array, so overflow can be detected; we make use of heap memory when node array is allocated.

Let me introduce our data structure more throughly. Code explanation.

C:\Users\R912COM\Downloads\structrgb.PNG

This is the base structure for the 2 dimensional image array, which consists of 3 unsigned char values, which represents the bit value of red, green and blue. As we only use 24-bit images, one pixel has 24-bit data, which is 8 bit red, 8 bit green and 8 bit blue.



This is the base structure for the node for the tree. This data is chosen by enum node\_status, which means NO\_NODE: No child node (Is the leaf node), LR\_NODE: Only has two nodes which is left and right node, UD\_NODE: Only has two nodes which is up and down, and finally ALL\_NODE: Has four complete nodes. The existence of LR\_NODE or UD\_NODE is because not every image is shaped square. So it is inevitable that irregular node such as 2x1 or 1x4 shows up. Those two types are for that exceptions. If it is a leaf node(NO\_NODE), we choose color value from the union, or we choose the child node array. It is obvious because leaf node does not have child nodes, and nodes who are not leaf nodes does not need to contain color information, because the color is not inconsistent, so it is divided into child nodes.

Also, there are four integer data, row, column, height and width, which is represented by their first character, r,c,h and w. r and c is the leftmost, upmost point of the 2 dimensional array. and h and w means the consistent data size from the r, c.



This function makes the tree recursively from 2 dimensional image array. This function first checks whether this designated node is made of pixels of the same gray-scale value. If it is, then it is the leaf node(the base case for the recursion), so save the color data and return the node, as this structure copies by value. Else, then the function allocates the child nodes, and then call the function for them one by one, with different r,c,h and w.



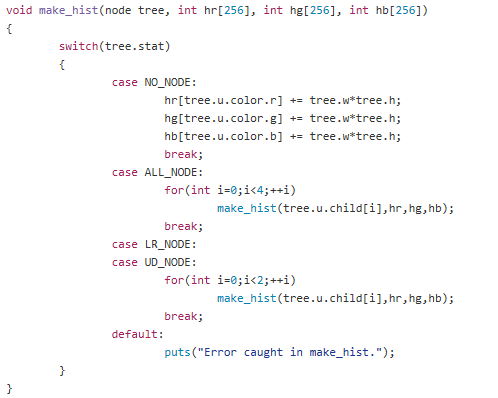
This function overwrites the 2 dimensional image array by the data from the tree. Every function after this is actually quite simple, as we made our tree as an object itself, not the pointer for the object, so every function has a switch case statement, which checks whether the node is a leaf node, or it has 2 or 4 children nodes, which is to be recursively called. In this case, this function overwrites the data of the 2-dimensional array from the data of the tree.

**4. Image Processing Implementation in Quadtree**

Note that image processing implementation in 2 dimensional array is quite simple, and we already described the algorithm fairly well in progress report, so we skip the 2 dimensional array representation.

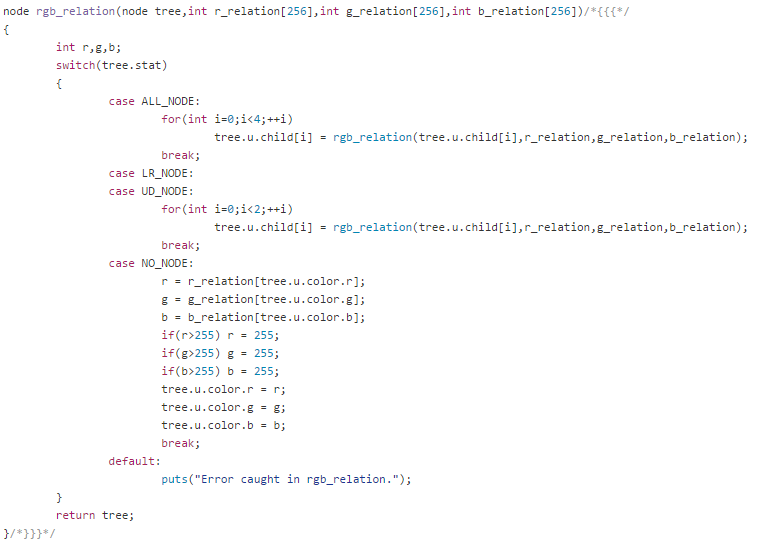
**Operations Using Histograms**

Histogram equalization, as explained above, consists of three key calculations. Making a histogram, calculating meaningful relation between color key, and then saving the calculated data. make\_hist() function makes the histogram recursively, then in the hist\_eq() function we change this histogram into a relation between two color values, then call function rgb\_relation() to save the data after applying relation.



As shown from 3. Data Stucture Implementation, this function also is recursively called. The calculation is done in the leaf node, which is adding the width and height of the leaf node, as the leaf node consists of width\*height pixels which has the same color.

After calculating the histogram with the above function make\_hist(), this function calculates the cumulative value of the histogram and then makes a relation between the before and after color value. As color value only has minimum 0, and maximum 255, relation function can be represented as a fixed array, size of 256. Then function rgb\_relation() is called.

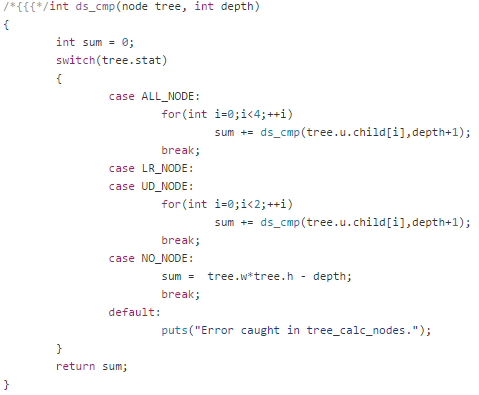


This function uses relation array input from parameter. It substitutes the result value from the original value to what that leaf node had.

**Operations Using Masks**

Mask operations are done solely by the function mask\_tree(). The resulting varying value are made by the input 3x3 matrix, which is usually called a kernel or a mask. This function uses both data structures, which is fixed array and quadtree. This function is similar with the original fixed array implementation, but it can jump over the unnecessary calculations using quadtree.

This function saves the precalculated color value in the RGB jumpresult[] array. As quadtrees saves color segments as square, the unnecessary calculations are shaped square in the image array. So we used two more integer arrays, int jumpcnt[] and int jumpsize[]. jumpsize[] array saves the jumpable calculations(which is, matrix calculation) and jumpcnt[] array saves the remaining jumps.



We determined whether to use quadtree or 2 dimensional array by the function ds\_cmp(). This function analyze the tree. Basically, for 2 dimensional array, traversing the whole node takes O(WIDTH\*HEIGHT) time, and for trees, O(TOTAL\_DEPTHS+NODES) time. TOTAL\_DEPTHS is the sum of each nodes’ depth.

So if WIDTH\*HEIGHT-TOTAL\_DEPTHS are positive, then it is better to use quadtrees rather than 2 dimensional array. The function ds\_cmp recursively calculates the result and returns the value.

**5. Technology Analysis**

This is developed primarily in C++, and used basic headers, namely, cstdlib, cstdio and cstring. Our base operating systems were Mac OSX or Ubuntu. So, the codes are made with vim editors(which indicates newline feed follows standard vim), and are optimized in GNU compiler collections, such as g++. However, as we closely follow ANSI C standards, it is guaranteed to run in other compilers just as well.

If we change the line system(“clear”) into system(“cls”), it is guaranteed to run on Windows-based system such as Visual Studio.

**6. Development Cycle Analysis**

The tools we used in working on the project are mainly Google Drive, Slack, and Github. Because all of the group members were new to using Github, we only uploaded bug-free codes to branches, and discussed the ideas, pseudo codes or first-draft codes in Slack, as Slack offers great file administrative system. Every reference we have used and the documentation of our project is uploaded and handled using Google Drive. Besides providing a large internet-based cloud storage, Google Drive gives multiple users the access to the large-size pdfs at the same time, so all three members of the group can simultaneously work on the same documentation.

We started with idea sharing at May 11th, and we pondered final topics for a week. We fixed our topic into image processing in C at May 24th, and analyzed the image files, such as bmp image file. We used 2 dimensional arrays for basical image operations before the progress report, and developed an implementation for quadtrees, program user interface, and comments after the report. We handled various exceptions throughout the whole period.

**7. Conclusion**

* Usage of different data structures
  + Choosing the data structure after analyzing the image
* Implementation of basic image operations in Quadtrees
  + Operations using masks (Edge detection, Sharpening & Blurring, etc.)
  + Operations using histograms (Equalization, Filtering, Segmentation, etc.)
* Practice of agile software development using Github and Google Drive