Header File for new class:

/\*

Container.h

Author: Cheuk-Hang Tse

Purpose: The header file for new Container class created for Program4

\*/

#pragma once

#include <iostream>

#include <vector>

#include "ImageLib.h"

#include "ImageClass.h"

// Node structure: include information about position and color components

// Also include the neighbor relation and is the Node visited(processed).

struct Node

{

int red;

int green;

int blue;

int row;

int col;

Node \*abovePixel = nullptr;

Node \*belowPixel = nullptr;

Node \*leftPixel = nullptr;

Node \*rightPixel = nullptr;

// Maybe use pointer array? with the size of 4?

// std::vector<Node> neighbor;

bool visited = false;

Node nextContainer = nullptr;

};

class Container

{

public:

// Constructor: construct an empty container.

// Precondition: None.

// Postcondition: Construct an empty container with head points to Node of row 0 and col 0.

Container();

// Copy Constructor: construct an exact copy of the input container.

// Precondition: None.

// Postcondition: Construct an exact copy of the input container and same head pointer.

Container(const Container& inContainer);

// operator=: deallocate memory in original container and copy exactly from input container.

// Precondition: \*this is not equal to inputContainer.

// Postcondition: The new container should be exactly the same as input container.

const Container& operator=(const Container& in);

// Deconstructor: deallocate memory in the container.

// Precondition: None.

// Postcondition: Deallocate all the memory in the container.

~Container();

// addPixel: add a node to the container based on the components of input pixel.

// Precondition: Need to check if color component in input pixel is valid.

// Postcondition: add the inputNode to the container based on the color component of inputPixel.

void addPixel(Node \*& head, Node \*&inputNode, int red, int green, int blue);

// merge: merge the mergeContainer to current container.

// Precondition: all node from current container and mergeContainer’s visited is reseted.

// Postcondition: connect head Node from mergeContainer to current container's nextContainer.

void merge(Container mergeCon);

// point the nextContainerHead to mergeCon.head

// isVisited: return true if the input Node is already visited

// Precondition: None.

// Postcondition: Return true if in->visited = true, else return false.

bool isVisited(Node in);

// resetVisited: Reset the visited for every Node in head linked list.

// Precondition: None.

// Postcondition: All node in the linked list, it's visited is reseted to false.

void resetVisited(Node \*& head);

// averageColor: average the color in the linked list in head.

// Precondition: visited is reset for all node.

// Postcondition: average the color in the linked list and

// add to the reference of red, green, blue. And return the number of pixel.

int averageColor(Node \*& head, int& red, int& green, int& blue);

// Traverse the linked list by while loop. Every node traversed, increase the number of pixel by 1.

// add the color to a red, green, and blue placeholder

// divide the red, green, and blue placeholder by number of pixel.

// add the red, green, blue values to red, green, and blue reference.

private:

// Store the start node on a group.

Node\* head = nullptr;

Node nextContainerHead = nullptr;

};

Driver:

//-------------------------------------------------------------------------------------//

// In Driver:

// formContainerBySeed: change this->container to a container that store all node in an Image

// Preconditions: Completed Node graph construction. head is pointed to row 0, col 0

// Postconditions: initial container will modified by merging multiple merge container during recursion

void formContainerBySeed(Node \*&head, const pixel& seed, Container& currentContainer);

// Recursion function:

// If head is not null or head is visited then exit recursion.

// Check if head pixel is "close" to seed pixel

// If head pixel is "close" to seed pixel, add head pixel to container

// Then, I would loop through neighbor pixel by recure by passing in neighbor node, seed, current container.

// formContainer(neighborNode, seed, currentContainer);

// Until the seed pixel is not close to head pixel, then I would

// create a new container call merge container and a new seed pixel by this head.

// Then recure by passing the head, new seed, and merge container.

// formContainer(head, newSeed, mergeContainer);

// At the end of "not close to head pixel" nest, I would merge the merge container to current container.

//-------------------------------------------------------------------------------------//

// createGraph: create an initial container that include a complete graph of pixel from inputImage

// Precondtions: Must be an image that at least have 1 \* 1 size. And head must be at row 0 and col 0.

// Postconditions: Return a container that include a complete node graph of the image's pixels.

void createGraph(ImageClass& in, Container& container, Node \*& head);

// Recursion function:

// Run by allocating pixel on head's row and col.

// Assign color component to the head and push into the container.

// If head's neighbor is nullptr and row + 1 or col + 1 is not out of range, and

// check that no added node in container that have determined location. Then,

// create a new node with determined row and col.

// neighborNode = new Node();

// neighborNode->row = row; and neighborNode->col = col;

// assign neighbor relationship.

// head->neighborDirection = neighborNode;

// neighbornode->headDirection = head;

// Then recure by passing the new node as head.

// createGraph(in, container, neighborNode);

//-------------------------------------------------------------------------------------//

// createImageByContainer: return an Image of the node from the container.

// Precondition: container have already processed by formContainerBySeed, averageColor, resetVisited.

// Postcondition: create an Image based on the Nodes in container

ImageClass createImageByContainer(Container container, const ImageClass& in);

// create an empty Image by passing in row and column parameter from inputImage

// create an empty pixel with color components equal to 0.

// Traverse through the linked list.

// Each Node in the list, set the pixel at location node.row and node.col

// to the empty pixel, which would have it's red, green, blue components equal to

// node.red, node.green, node.blue

Important function will be used in this program:

From my ImageClass:

// getRows: Return the rows of this Image class.

// Preconditions: coreImage exists in this class.

// Postconditions: Return the rows of the coreImage in the class.

int getRows() const;

// getColumns: Return the columns of this Image class.

// Preconditions: coreImage exists in this class.

// Postconditions: Return the columns of the coreImage in the class.

int getColumns() const;

// Return a pixel of this Image class.

// Preconditions: Must have valid row and column inputed.

// Postconditions: return a pixel from the coreImage at location (row ,column).

const pixel& getPixel(const int& row, const int& col);

// Set a pixel of this Image class.

// Preconditions: Must have valid row and column inputed.

// Postconditions: Set the coreImage's a pixel at location (row ,column) to inputPixel.

void setPixel(const int& row, const int& col, const pixel& inputPixel);

// Constructor using number of rows and columns

// Preconditions: None.

// Postconditions: Constructe an empty image with the parameters: rows and cols.

ImageClass(const int& rows, const int& cols);

// outputImage: Output a GIF file with the filename.

// Preconditions: coreImage exist in the class.

// Postconditions: Write the coreImage as a GIF file with the filename.

const void outputImage(std::string& filename);

// Use WriteGIF