Individual Homework: Seam Carving (Part 1)[[1]](#footnote-0)

# Objectives

* Further practice reading from files.
* Validation of input data including the use of stream states.
* Work with two-dimensional arrays.
  + Traversing through.
  + Accessing elements.
  + Staying within array bounds.
* Work with dynamic arrays.
* Compute information based on information from different parts of an array.

# Submission

* seamcarving.cpp
* functions.h
* functions.cpp

# Overview

Sometimes you want to use an image, but it needs to be resized for your use. For example, what if we need to make this image narrower without changing the height?

Original surfer image.  


Traditionally this is done by cropping or re-scaling.

Cropping can remove important information such as the two people on the right of the image.

Cropped surfer image.  
Island to left and surfer are clearly visible. Only one surfer from the cluster on the right is visible.


Rescaling ends up making the image look squashed.

Rescaled surfer image.  


Something that would be ideal is a way to resize the image that does not distort the important details of the image. Seam carving is a method of resizing images in a content-aware manner that works well for some types of images. Unimportant pixels are removed while pixels that convey important details are preserved. Interestingly, the seam carving approach was first published in 2007[[2]](#footnote-1).

[Watch the original SIGGRAPH video describing the technique](https://www.youtube.com/watch?v=6NcIJXTlugc).

Seam carved surfer image.

Download zip of test ppm files

# Requirements

* [Get the start code](https://drive.google.com/drive/folders/1xSiCzMhuYYvr7B1joEh4SHx6ZY7999tT?usp=sharing)..
  + Files
    - seamcarving.cpp
      * Update as needed
    - functions.h
      * Update as needed (you probably won’t need to)
    - functions.cpp
      * Update as needed.
        + If you add more functions, be sure to add their declarations to functions.h.
* Download test ppm files
  + Download [zip of test ppm files](https://drive.google.com/file/d/1-yDgNIh5jBVRJmIuKS5XYq6OOvLVB_Ig/view?usp=sharing)
  + See [test ppm files on Google Drive](https://drive.google.com/drive/folders/12JYchp76EAxu5RF0KL0c58iLIGksyj8x?usp=sharing)
    - View these files to see what they look like.
    - As you test your program, view the modified ppm files to see how well your solution is working.
    - Bigger files will take longer to run, especially later in part 2.
* You may only include
  + <iostream>
  + <fstream>
  + <string>
  + <sstream>
  + <cmath>
  + “functions.h”
* Read over the starting code.
  + The struct Pixel is defined in functions.h
  + The main program already reads in a file name, the width and height, and the target width and height after carving. You can use that to test your program.
  + You will need to use an image viewer to view your PPM file. See “[Viewing ppm files](#_3fn2fi580pf)” section below for information on viewing your PPM files.
  + We are using **dynamic arrays** for this homework.
* Update seamcarving.cpp to error check user input of width, height, targetWidth, and targetHeight.
  + When an error is encountered, exit the program immediately with an error after printing an error message.
  + Print error messages to standard out
    - Error: <valueName> must be greater than 0. You entered <value>
      * When width, height, targetWidth or targetHeight are less than or equal to 0.
        + E.g. Error: target width must be greater than 0. You entered 0
    - Error: <targetValueName> must be less than <valueName>, <targetValue> is greater than <value>
      * When targetWidth is greater than width or targetHeight is greater than height
        + E.g. Error: target width must be less than width, 88 is greater than 50
    - Error: <targetValueName> is a non-integer value
      * When trying to read an integer, but got something that is not valid for an integer.
        + E.g. Error: target width is a non-integer value
* Implement **first**  
  bool loadImage(const char filename[],   
   Pixel\*\* image, int width, int height);
  + This function will open the PPM file and load the 2d image array with pixels using RGB values from the file.
    - See “[Images and RGB Color Model](#_hvrvjjmu023w)” section below for a description of images, the RGB color model, and the PPM file format.
  + The function returns true if loading the file was successful and false otherwise
  + The first parameter is the name of the file to open for input
  + The second parameter is a 2d array of Pixels (structs) that hold a color value
  + The third parameter is the width of the array (i.e. the number of columns) needed for traversing the array
  + The fourth parameter is the height of the array (i.e. the number of rows) needed for traversing the array.
  + The array should be filled in column-major order. See “[Images and RGB Color Model](#_hvrvjjmu023w)” section below for a detailed explanation.
  + Part of this function is already written
  + The function must include error checking.
    - When an error is encountered, return false and immediately return from the function.
      * Print error messages to standard out.
      * Error: failed to open input file - <filename>
        + When you fail to open the input ppm file
      * Error: type is <type> instead of P3
        + When the ppm file type does not match the P3 or p3 that we expect.
      * Error: read non-integer value
        + When trying to read an integer, but got something that is not valid for an integer.
      * Error: input <valuename>(<inputVal>) does not match value in file (<fileVal>)
        + When a value typed into the program do not match the corresponding value in the ppm file.

E.g. Error: input width (77) does not match value in file (72)

* + - * Error: invalid color value <badValue>
        + When reading a color value and get a value that is less than 0 or greater than the max color value (e.g. 255).
      * Error: not enough color values
        + When there are not enough numbers for all colors in all pixels.
      * Error: too many color values
        + When there are more numbers than needed for all of the pixels in the image.g
* Implement **second**  
  bool outputImage(const char filename[],   
   const Pixel\*\* image, int width, int height);
  + This function will output a two-dimensional image array of pixels to a PPM file.
    - See “[Images and RGB Color Model](#_hvrvjjmu023w)” section below for a description of images, the RGB color model, and the PPM file format.
    - Always output 255 for the max color value
  + The function returns true if saving the file was successful and false otherwise
  + The first parameter is the name of the file to open for output.
  + The second parameter is a 2d array of Pixels (structs) that hold a color value
  + The third parameter is the width of the array needed for traversing the array
  + The fourth parameter is the height of the array needed for traversing the array.
  + Make sure you include the preamble and put spaces between numbers.
    - It is useful to get output working before you try and modify the image. So make sure this is working before you start on the functions that modify the image.
  + Print error messages to standard out.
    - Error: failed to open output file - <filename>
      * When you fail to open the output ppm file.
* Implementint energy(const Pixel\*const\* image, int column, int row, int width, int height);
  + This function will return the dual-gradient energy of the pixel. How to calculate this is in the “[Computing the energy of a pixel](#_hsvef6jluhbk)” section below.
  + The first parameter is a pointer to a pointer to a Pixel. With this structure, we can access our array with the [][] notation. Since we are using dynamic arrays, the width is not standard, so we cannot use [][] in the function signature since we would have to specify the array dimensions at compile-time, but we get the dimensions (e.g. width) at run time.
  + The second parameter column is the column which is the first value in the coordinate (column, row).
  + The third parameter row is the row which is the second value in the coordinate (column, row).
  + Note you can write a function that prints out a table of the energy for each pixel coordinate. This is a great debugging tool for seeing if your energies are correct.
* Implement  
  int\* createSeam(int length);
  + This function will return a pointer to an array of ints allocated on the heap. The array must be initialized so that all elements have a value of zero in them.
  + The parameter indicates how many elements must be in the returned array. You can also think of this as the size of the array
* Implement (after implementing createSeam)  
  void deleteSeam(int\* seam)
  + This function will deallocate the memory for the indicated array, returning its memory to the heap.
  + The parameter is a pointer to an array of ints on the heap.

# Supporting Information

## Images and RGB Color Model

Images are a two-dimensional matrix of pixels where each pixel is a color composed of a red, a green, and a blue value. For example RGB(80, 0, 0) is Aggie maroon.[[3]](#footnote-2)

In image processing, pixel (x,y) refers to the pixel in column x and row y where pixel (0, 0) is the upper left corner of the image and pixel (width-1, height-1) is in the lower right corner.

***Warning:*** This is column-major ordering which is transposed from the row-major ordering that is used for cartesian coordinates where the first index is the row and the second index is the column and (0, 0) is in the lower-left corner. In image files, the width is essentially the number of columns and the height is the number of rows. So the impact is that you will index with [col][row] instead of with [row][col].

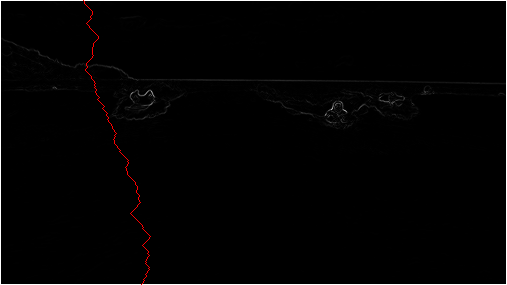
Coordinates for a 3X4 (3 columns (i.e. width) by 4 rows (i.e. height)) image are shown in the following table.

| (0, 0) | (1, 0) | (2, 0) |
| --- | --- | --- |
| (0, 1) | (1, 1) | (2, 1) |
| (0, 2) | (1, 2) | (2, 2) |
| (0, 3) | (1, 3) | (2, 3) |

We will use a Pixel struct (defined in functions.h) that holds a value for red, green and blue. The image will be a 2 dimensional array of Pixels.

## Seam Carving

Seam carving involves three major steps.

1. **Energy Calculation** (You’ll do this week in Part 1)  
   The first step is to calculate the energy of a pixel, which is a measure of its importance—the higher the energy, the less likely that the pixel will be included as part of a seam (as we'll see in the next step). In this assignment, you will use the dual-gradient energy function, which is described in the “[Computing the energy of a pixel](#_hsvef6jluhbk)” section below.  
   Here is a visualization of the dual gradient of the surfing image above.  
   The energy is high (white) for pixels in the image where there is a rapid color gradient (such as the boundary between the sea and sky and the boundary between the surfing Josh Hug on the left and the ocean behind him). The seam-carving technique avoids removing such high-energy pixels.
2. **Seam Identification** (You’ll do next week in Part 2)  
   The next step is to find a vertical seam of minimum total energy.   
   Seams cannot wrap around the image (e.g., a vertical seam cannot cross from the leftmost column of the image to the rightmost column).  
   Finding a horizontal seam is analogous.
3. **Seam Removal** (You’ll do next week in Part 2)  
   The final step is remove from the image all of the pixels along the vertical or horizontal seam.

### Computing the energy of a pixel

Recall the notation covered above in “[Images and RGB Color Model](#_hvrvjjmu023w)”

You will use the *dual-gradient energy function*: The energy of pixel (*x*, *y*) is Δ*x*2(*x*, *y*) + Δ*y*2(*x*, *y*), where the square of the *x-*gradient Δ*x*2(*x*, *y*) = *Rx*(*x*, *y*)2 + *Gx*(*x*, *y*)2 + *Bx*(*x*, *y*)2, and where the central differences *Rx*(*x*, *y*), *Gx*(*x*, *y*), and *Bx*(*x*, *y*) are the absolute value in differences of red, green, and blue components between pixel (*x* + 1, *y*) and pixel (*x* − 1, *y*). The square of the *y*-gradient Δ*y*2(*x*, *y*) is defined in an analogous manner. To handle pixels on the borders of the image, calculate energy by defining the leftmost and rightmost columns as adjacent and the topmost and bottommost rows as adjacent. For example, to compute the energy of a pixel (0, *y*) in the leftmost column, we use its right neighbor (1, *y*) and its left neighbor (*width* − 1, *y*).

Consider the 3-by-4 image with RGB values (each component is an integer between 0 and 255) as shown in the table below.

| (0, 0): (255, 101, 51) | (1, 0): (255, 101, 153) | (2, 0): (255, 101, 255) |
| --- | --- | --- |
| (0, 1): (255, 153, 51) | (1, 1): (255, 153, 153) | (2, 1): (255, 153, 255) |
| (0, 2): (255, 203, 51) | (1, 2): (255, 204,153) | (2, 2): (255, 205, 255) |
| (0, 3): (255, 255, 51) | (1, 3): (255, 255,153) | (2, 3): (255, 255, 255) |

* *Non-border pixel example.* The energy of pixel (1, 2) is calculated from pixels (0, 2) and (2, 2) for the *x*-gradient  
  *Rx*(1, 2) = 255 − 255 = 0,  
  *Gx*(1, 2) = 205 − 203 = 2,  
  *Bx*(1, 2) = 255 − 51 = 204,  
    
  yielding Δ*x*2(1, 2) = 02 +  22 + 2042 = 41620;  
    
  and pixels (1, 1) and (1, 3) for the *y*-gradient  
  *Ry*(1, 2) = 255 − 255 = 0,  
  *Gy*(1, 2) = 255 − 153 = 102,  
  *By*(1, 2) = 153 − 153 = 0,  
    
  yielding Δ*y*2(1, 2) = 02 + 1022 + 02 = 10404.  
    
  Thus, the energy of pixel (1, 2) is 41620 + 10404 = 52024. Similarly, the energy of pixel (1, 1) is 2042 + 1032 = 52225.
* *Border pixel example.* The energy of the border pixel (1, 0) is calculated by using pixels (0, 0) and (2, 0) for the *x*-gradient  
  *Rx*(1, 0) = 255 − 255 = 0,  
  *Gx*(1, 0) = 101 − 101 = 0,  
  *Bx*(1, 0) = 255 − 51 = 204,  
    
  yielding Δ*x*2(1, 0) = 02 + 02 + 2042 = 41616;  
    
  and pixels (1, 3) and (1, 1) for the *y*-gradient  
  *Ry*(1, 0) = 255 − 255 = 0,  
  *Gy*(1, 0) = 255 − 153 = 102,  
  *By*(1, 0) = 153 − 153 = 0,  
    
  yielding Δ*y*2(1, 0) = 02 + 1022 + 02 = 10404.  
    
  Thus, the energy of pixel (1,0) is 41616 + 10404 = 52020.

Table of all pixel energies for the RGB sample shown above.

| 20808 | 52020 | 20808 |
| --- | --- | --- |
| 20808 | 52225 | 21220 |
| 20809 | 52024 | 20809 |
| 20808 | 52225 | 21220 |

*Hint: Make a ppm file from the RGB colors above. Create a loop that calculates the energy of each pixel and see if it matches the energies above!*

## Image File Format (PPM)

You are probably already familiar with common image formats such as JPEG, PNG, and GIF. However, these formats all use some type of data compression to keep file sizes relatively small. However, we are not ready to tackle these formats in C++.

We are going to use an image format that only requires basic text file I/O.

[The PPM (portable pixel map) format](http://netpbm.sourceforge.net/doc/ppm.html) is a specification for representing images using the RGB color model. PPM is not used widely because it is very inefficient (for example, it does not apply any data compression to reduce the space required to represent an image.) However, PPM is very simple, and there are programs available for Windows, Mac, and Linux that can be used to view ppm images. Even more conveniently, you can use an online tool with your browser to [view your PPM files online](http://paulcuth.me.uk/netpbm-viewer/) or convert it into a widely supported format such as JPG. We will be using the [plain PPM version](http://netpbm.sourceforge.net/doc/ppm.html#plainppm), which stores the data in ASCII (i.e. plain text) rather than in a binary format. Since it is plain text, we will be able to use text file I/O to read and write these image files.

Note that the pixels in a PPM file are given row by row, so is essentially row-major ordering which is transposed from the array image format which is column-major.

If you do create your own plain / ASCII PPM files make sure you **remove the comments**, since we are not addressing how to identify and ignore comment lines. Comments are lines that start with the ‘#’ character. I used [the GIMP](https://www.gimp.org/) to create the PPM files provided with the starting code.

### PPM File Specification

* Preamble
  + First line: string “P3”
  + Second line: width (number of columns) and height (number of rows)
  + Third line: max color value (for us, 255)
* Rest of the file: list of RGB values for the image, expressed as a raster of rows, from top to bottom. Each row contains the RGB values (i.e., three values) for each column.

### PPM Examples

***Note:*** We have added colors to emphasize that every three numbers represent a single pixel. This version has each row on a separate line.

| P3 4 4 255 0 0 0 255 0 0 0 0 0 0 255 0  255 255 255 255 0 255 0 0 0 0 255 0  255 255 0 0 0 255 125 0 255 255 0 125 0 0 255 255 255 0 125 125 125 239 239 239 |
| --- |

This version is same as above, but with spaces added to help you visualize the file.

| P3 4 4 255  0 0 0 255 0 0 0 0 0 0 255 0  255 255 255 255 0 255 0 0 0 0 255 0  255 255 0 0 0 255 125 0 255 255 0 125  0 0 255 255 255 0 125 125 125 239 239 239 |
| --- |

This version has all numbers on a single line.

| P3 4 4 255 0 0 0 255 0 0 0 0 0 0 255 0 255 255 255 255 0 255 0 0 0 0 255 0 255 255 0 0 0 255 125 0 255 255 0 125 0 0 255 255 255 0 125 125 125 239 239 239 |
| --- |

Alternatively, it could be saved with one pixel per line (i.e. 3 numbers per line) or even one number per line (this is what the GIMP does when it creates PPM files).

The following also works, but makes no sense to a human reading it.

| P3 4 4 255 0 0 0 255 0 0  0 0 0 0 255  0 255 255 255 255  0 255 0 0 0  0 255 0 255 255  0 0 0 255 125  0 255 255 0 125  0 0 255 255 255  0 125 125 125 239  239 239 |
| --- |

Sample PPM File: [blocks.ppm](https://drive.google.com/file/d/1HlxOo1oh9lyiQ_etWnPefMarQ-DWquBq/view) Image for blocks.ppm file. Various sized non overlapping rectangles on a white background. A black rectangle in the upper left quadrant, a blue rectangle in the upper left quadrant, a yellow rectangle in the lower left quadrant, and a red rectangle in the lower right quadrant.

### Viewing PPM files

You’ll need to view your PPM files to see the results of your program. Unfortunately, PPM is not supported by many image viewers.

Some options for viewing your files include:

* [Drag files onto this website](http://paulcuth.me.uk/netpbm-viewer/) (http://paulcuth.me.uk/netpbm-viewer/)
  + You don’t have to download any programs!
* [The GIMP](https://www.gimp.org/) is an open source version of Photoshop.
  + ***Warning:*** This is a very large program.
  + If you use the GIMP to create any PPM files, you will need to remove the comment line that it adds (i.e. lines that starts with #).
* For Windows users, [IrfanView](https://www.irfanview.com/) is a free image viewer
  + Check the image-only box when installing.
  + Consent to allow IrfanView to associate to your image files.
  + After completing the installation, the image can be viewed by double-clicking on the PPM file.

1. [Seam carving assignment source: http://nifty.stanford.edu/2015/hug-seam-carving/](http://nifty.stanford.edu/2015/hug-seam-carving/) [↑](#footnote-ref-0)
2. Shai Avidan and Ariel Shamir. 2007. Seam carving for content-aware image resizing. ACM Trans. Graph. 26, 3, Article 10 (July 2007). DOI: https://doi-org.srv-proxy2.library.tamu.edu/10.1145/1276377.1276390 [↑](#footnote-ref-1)
3. [TAMU Web Color Pallette (https://brandguide.tamu.edu/web/web-color-palette.html)](https://brandguide.tamu.edu/web/web-color-palette.html) [↑](#footnote-ref-2)