Seam Carving (Part 1)

Overview

Objectives

- Using input/output streams
- Using file streams
- Validation of input data including the use of stream states
- Throwing exceptions.
- Work with two-dimensional arrays.
    - Traversing through
    - Accessing elements
    - Staying within array bounds
- Compute information based on information from different parts of an array

Introduction

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

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[1]. Watch the original SIGGRAPH video describing the technique.

Seam carved surfer image.

1 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

Getting Started

- Get the starter code (same as what you submit).
  - `seamcarving.cpp`
    - Update as needed.
  - `functions.h`
    - Will not need to update for part 1.
  - `functions.cpp`
    - Update as needed.
    - If you add more functions, be sure to add their declarations to functions.h.
- Download ppm files.
  - 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.
- Allowed includes:
  - `<iostream>`
  - `<fstream>`
  - `<string>`
  - `<sstream>`
  - `<cmath>`
  - `"functions.h"`
- Read over the starting code.
  - The `struct Pixel` is defined in `functions.h`
  - The maximum dimensions of the image are defined in `functions.h` (`MAX_WIDTH` and `MAX_HEIGHT`).
  - The main program already initializes a 2D array (`image`), reads in a filename, calls `loadImage()`, reads in target dimensions, and calls `outputImage()`.
  - See "viewing ppm files" section below for information on viewing your PPM files.
- Read the requirements.
  - Go through supporting information as needed.
- Complete seam carving part 2 (next week)

Requirements

The requirements are given in the approximate order they should be completed.

`functions.cpp`

You will need to implement three functions. Do not change the function prototypes (name, parameters, or return type). You should only write the bodies.

`loadImage()`

- **Description**: opens and validates a PPM file while populating an array of pixels.
  - Should read in the preamble and pixel values (you can use a file stream).
  - Should validate values as they are read in.
  - Should use stream states to ensure successful input.
  - PPM files are in **row major** order.
- **Parameters**:
  - `string filename`
    - Name of PPM file to read from.
  - `Pixel image[][MAX_HEIGHT]`
    - 2D array of pixels **to populate** (`MAX_WIDTH` x `MAX_HEIGHT`).
    - In **column major** order.
  - `unsigned int& width`
    - Width of image (**should be updated**).
  - `unsigned int& height`
    - Height of image (**should be updated**).
- **Returns**: `void`.
- **Exceptions**:
  - If the file cannot be opened, throw a `runtime_error` exception with the description `"Failed to open <filename>"`.
  - If the file type is not `"P3"` or `"p3"`, throw a `runtime_error exception` with the description `"Invalid type <type>"`.
  - Both dimensions should be positive integers and less than or equal to the maximum's defined in `functions.h`.
    - If the dimensions are invalid, throw a `runtime_error` exception with the description `"Invalid dimensions"`.
  - Each pixel should be 3 (red, green, and blue) non-negative integers less than 256.
    - If there is an invalid pixel value, throw a `runtime_error` exception with the description `"Invalid color value"`.
    - If there are not enough pixel values, throw a `runtime_error` exception with the description `"Invalid color value"`.
    - If there are too many pixels, throw a `runtime_error` exception with the description `"Too many values"`.

`outputImage()`

- **Description**: Outputs an array of pixels to a PPM file.
  - Should output the preamble and all the pixel values (separated by spaces / newline characters). You can use a file stream.
  - PPM files are in **row major** order.

- **Parameters**:
  - `filename`
    - Name of PPM file to output to.
  - `Pixel image[][MAX_HEIGHT]`
    - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`).
    - In **column major** order.
  - `unsigned int width`
    - Width of image.
  - `unsigned int height`
    - Height of image.
- **Returns**: `void`.
- **Exceptions**:
  - If the output file cannot be opened, throw a runtime_error exception with the description `"Failed to open <filename>"`.

`energy()`

- **Description**: Calculates the energy of a specific pixel in an array array.
  - See the computing the energy of a pixel section for specifics.
- **Parameters**:
  - `Pixel image[][MAX_HEIGHT]`
    - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`).
    - In **column major** order.
  - `unsigned int x`
    - X coordinate (column) of pixel.
  - `unsigned int y`
    - Y coordinate (row) of pixel.
  - `unsigned int width`
    - Width of image.
  - `unsigned int height`
    - Height of image.
- **Returns**: unsigned `int`.
  - The calculated energy.
- **Exceptions**: none.

seamcarving.cpp

- Validate `targetWidth` and `targetHeight` from user input.
  - See `"TODO: add code to validate input (part 1)"` in `main()`.
  - Both dimensions should be positive and less than the maximum's defined in `functions.h`.
  - If the dimensions are invalid, output `"Invalid target dimensions"` and exit with an error.

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.[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

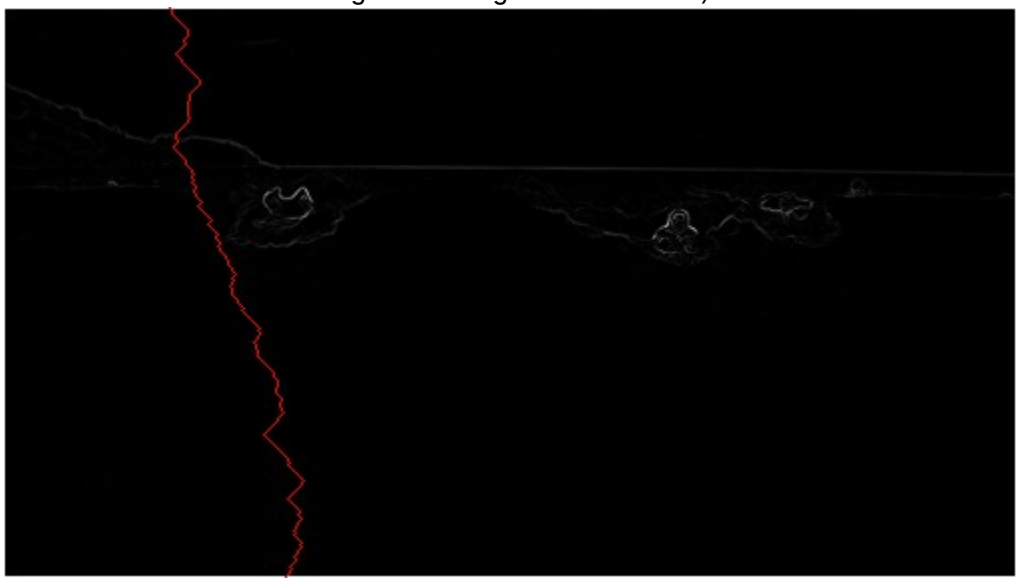
Seam carving involves three major steps.

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" section below. Here is a visualization of the dual gradient of the surfing image above.

2 TAMU Web Color Pallette (https://brandguide.tamu.edu/web/web-color-palette.html)

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 to remove from the image all of the pixels along the vertical or horizontal seam.

Computing the energy of a pixel

You will use the *dual-gradient energy function*: The energy of pixel (x, y) is $\Delta_x^2(x, y) + \Delta_y^2(x, y)$, where the square of the x-gradient $\Delta_x^2(x, y) = R_x(x, y)^2 + G_x(x, y)^2 + B_x(x, y)^2$, and where the central differences $R_x(x, y)$, $G_x(x, y)$, and $B_x(x, y)$ 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 $\Delta_y^2(x, y)$ 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
  $R_x(1, 2) = 255 - 255 = 0,$
  $G_x(1, 2) = 205 - 203 = 2,$
  $B_x(1, 2) = 255 - 51 = 204,$

  yielding $\Delta_x^2(1, 2) = 0^2 + 2^2 + 204^2 = 41620$;

  and pixels (1, 1) and (1, 3) for the *y*-gradient
  $R_y(1, 2) = 255 - 255 = 0,$
  $G_y(1, 2) = 255 - 153 = 102,$
  $B_y(1, 2) = 153 - 153 = 0,$

  yielding $\Delta_y^2(1, 2) = 0^2 + 102^2 + 0^2 = 10404$.

  Thus, the energy of pixel (1, 2) is 41620 + 10404 = 52024. Similarly, the energy of pixel (1, 1) is $204^2 + 103^2 = 52225$.

- *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
  $R_x(1, 0) = 255 - 255 = 0,$
  $G_x(1, 0) = 101 - 101 = 0,$
  $B_x(1, 0) = 255 - 51 = 204,$

  yielding $\Delta_x^2(1, 0) = 0^2 + 0^2 + 204^2 = 41616$;

  and pixels (1, 3) and (1, 1) for the *y*-gradient
  $R_y(1, 0) = 255 - 255 = 0,$
  $G_y(1, 0) = 255 - 153 = 102,$
  $B_y(1, 0) = 153 - 153 = 0,$

  yielding $\Delta_y^2(1, 0) = 0^2 + 102^2 + 0^2 = 10404$.

  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 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 or convert it into a widely supported format such as JPG. We will be using the plain PPM version, 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. The GIMP was used 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 0 255 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.

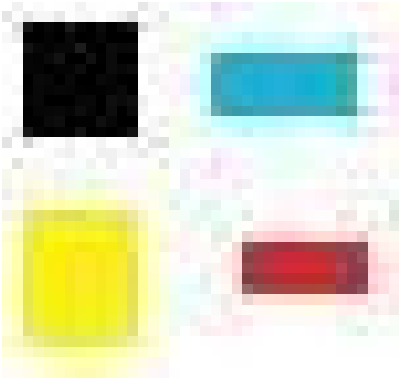
```
P3
4 4
255
0 0 0 255 0 0 0 0 0 0 255 0 255 255 255 255 0 255 0 0 0 0 255 0 255
255 0 0 0 255 125 0 255 255 0 125 0 0 255 255 255 0 125 125 125 239
239 239
```

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



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/)
  - You don't have to download any programs!
- The GIMP 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 any comment lines that it adds (i.e. lines that starts with #).
- Krita is a more lightweight, open source, illustrator program with PPM file support.
- For Windows users, IrfanView 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.