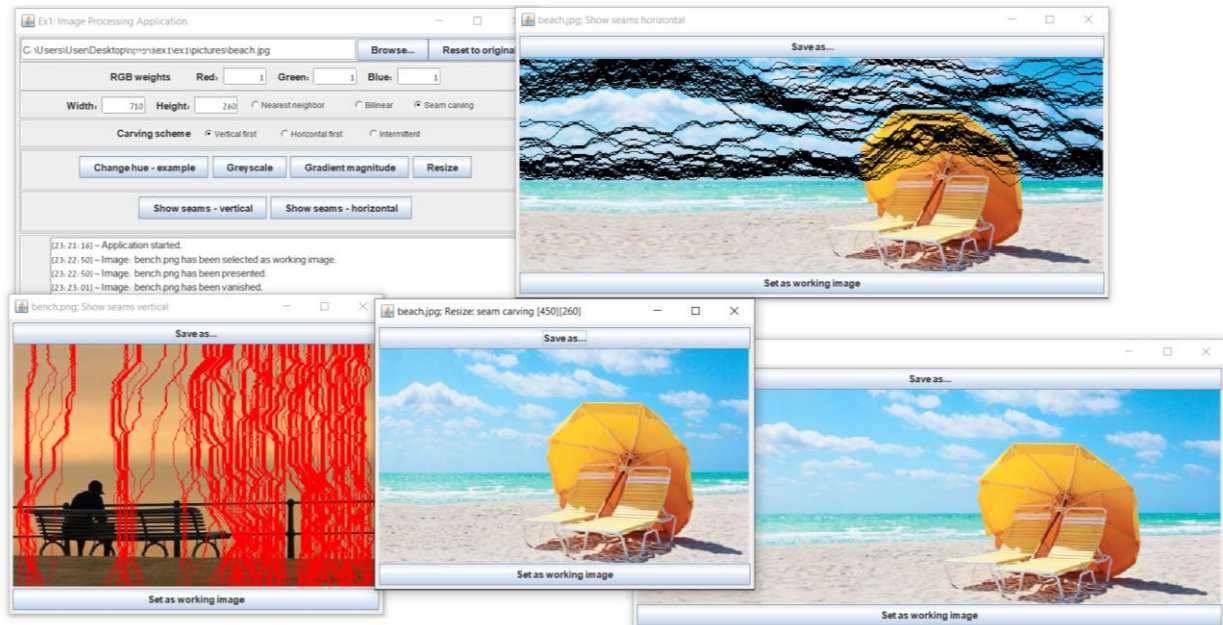# Computer Graphics ex1

The Interdisciplinary Center 2021
Prof. Ariel Shamir
Submission date: Monday April 6, 11:55pm

Image Processing and Seam Carving

In this exercise you will implement and explore image resizing using the seam carving algorithm, along with other basic image processing operations.

To illustrate the different operations you are provided with an example working java application that illustrates these operations:

- Changing the image's hue – already implemented as an example.
- Conversion into grayscale.
- Calculation of gradient magnitude.
- Resizing an image using three methods:
  - Nearest neighbor.
  - Bilinear interpolation.
  - Seam carving.

Specifically, we use three different carving schemes for seam carving :

Scheme 1 : Removing vertical seams first to resize the image's width, then removing horizontal seams from the resulting image.

Scheme 2: Removing horizontal seams first to resize the image's height, then removing vertical seams from the resulting image.

Scheme 3: Removing seams intermittently : vertical, horizontal, vertical, horizontal etc.

These schemes are discussed at great length below.

You are also provided with partial code, which you will need to complete to achieve an application that is similar to the given one.

What you should do

1. Install java (jdk) 11 or newer, if you are using macOs this video will guide you how to remove the older version of jdk.
2. Run ex1complete.jar, which is a binary similar to what you are going to implement, and play around.
    If double click doesn't work, you can run the next command line:
    ```
    java -jar ex1complete.jar
    ```
3. Read the partial code given in the src directory for this exercise. Understand what each class does and how.
4. You can use any kind of java IDE, but we will check you solution with IntelliJ, make sure it works in IntelliJbefore you submit.
5. Your responsibility is to fill in the missing pieces according to the functionality described below. **Implement** all TODO's in the code (search for the string "TODO" and replace it with your code). The final result should behave the same as ex1complete.jar.
6. Submit your implementation in a zip file according to the submission guidelines below.

Submission Guidelines

Submit the "src" folder with all the java source files, including the files you didn't change.
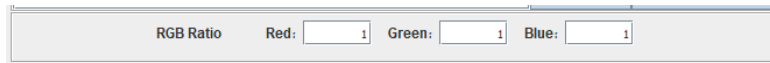Zip it to a file called:
```
<Ex##> <FirstName1> <FamilyName1> <ID1> <FirstName2> <FamilyName2> <ID2>
```
For example: `Ex01 Bart Cohen-Simpson 34567890 Darth Vader-Levi 12345678`
Upload the file to the Moodle site.

PART A – Basic image processing : overview

(You should complete this part before moving onto PART B)

- **Change hue – example:** This function is **already implemented**, explore its code. Note that it uses the functionalities offered by the class FunctioalForEachLoops. It may be convenient for you to use these functionalities in your code, but it is **not required**. You can use standard for loops to iterate over 2D arrays/Images.
- **Greyscale:** Convert the image into grayscale, using the RGB weights that was entered in the application main window:

	RGB Ratio	Red:	1	Green:	1	Blue:	1

You can change those values to get different results. (Each value should be an integer $\in$ [0,100] and the total amount should be greater than zero).

Use the next formula to calculate the grey scaled image:
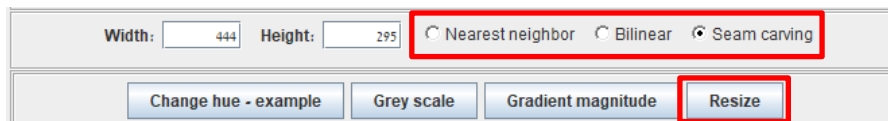
for each pixel p $\in$ input image do:

$$greyColor = \frac{p.red \times weights.red + p.green \times weights.green + p.blue \times weights.blue}{weights.red + weights.green + weights.blue}$$

set the greyColor as the output image's pixel.

- **Gradient magnitude:** The gradient magnitude should be computed as: $\sqrt{\frac{dx^2 + dy^2}{2}}$,

where dx is the difference between the current and next horizontal pixel, and dy is the difference between the current and next vertical pixel. Pay attention to the image borders conditions. Take the previous pixel instead of the next one. If the image dimensions are too small, throw an appropriate exception.

Note: The gradient magnitude is based on the grey scaled image by using the given RGB weights.

- **Resize:** By clicking the "Resize" button, an invocation of the selected method will be performed.

The methods are:

- Nearest neighbor - example: Each new sized image pixel's color will be taken from the nearest appropriate pixel in the original image. This is **already implemented**, review the code to get an impression of resizing an image.
- Bilinear interpolation: Each new sized image pixel's color will be calculated as a bilinear interpolation by using the four closest surrounding pixels of the original image. Each image has different dimensions. Think how to calculate the transformations. You can read more about bilinear interpolation in the course summaries that are on Moodle, or here.

- Seam carving : You will implement in PART B of this exercise.
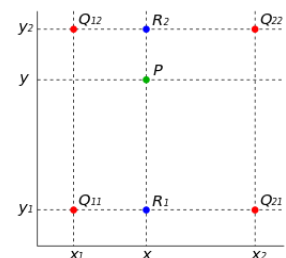
PART A – Basic image processing : code

The source consists of the following classes:

Main – The entry point to the program. It is fully implemented.

MenuWindow – A class that represents the app's main window. It is fully implemented.

ImageWindow – A window that displays an image and allows you to save it as a PNG file. It is fully implemented.

Logger – An interface providing the log method(s), to print messages to the log field in the app's main window.

FunctioalForEachLoops – An abstract class that provides a convenient way of iterating over 2D arrays (also 1D arrays). It is fully implemented.

RGBWeights – A class that represents the weight of each color channel.

ImageProcessor extends **FunctioalForEachLoops** – A class that has some image processing methods:

> `logger` – A Logger type field that prints messages to the log field in the app's main window. You can use it the way you want.

> `changeHue()` – This is already implemented as an example. Test the implementation.

> `greyscale()` - Makes an image greyscale. You should implement this. The rgbWeights field should affect the result.

> `gradientMagnitude()` - Calculates the magnitude of gradient at each pixel. You should implement this. The rgbWeights field should affect the result.

> `nearestNeighbor()` – Resize the image by using the nearest neighbor interpolation. This is **already implemented**, review the code to get an impression of resizing an image.

> `bilinear()` – Resize the image by using the bilinear interpolation. You should implement this.

PART B – Seam Carving : overview

First, let us recall and review the seam carving algorithm.

Seam Carving – algorithm review

The explanation here is discussed at great length in the recitation.

- Assume we need to carve k seams from an image.
- For each seam:
  - Calculate the costs matrix. The following formula is for vertical seam carving, you can find the complementary horizontal formula in the recitation.

$$M_{y,x} = pixelEnergy(y,x) + \min \begin{cases} M_{y-1,x-1} + C_L(y,x) \\ M_{y-1,x} + C_V(y,x) \\ M_{y-1,x+1} + C_R(y,x) \end{cases}$$

  - Use dynamic programming to find the optimal seam. First find the smallest cost in the bottom row (in the vertical case), then, travel back to the top along the path of minimal costs (invert the formula above).
  - Store the path of each seam in an appropriate data structure.
  - Remove the seam.
- To reduce the image's width/height by k, the seams that were chosen in this procedure need to be removed from the image.
- To increase the image's width/height by k, the seams that were chosen in this procedure need to be duplicated within the image.

Now, let us consider three different schemes for performing seam carving.

Seam Carving – carving schemes

According to the algorithm we need to remove/duplicate horizontal and vertical seams to achieve the desired image scale.

But at what order is this carried out? For example – If we need to remove n vertical seams and m horizontal seams, do we **first** remove n vertical seams and then remove m horizontal seams from the resulting image? Or, do we **first** remove m horizontal seams and then remove n vertical seams from the resulting image?

There are of course other options – we can for example remove one vertical seam, then remove one horizontal seam, then remove one vertical seam, then one horizontal etc. We can continue this intermittent removal until we have removed n vertical seams and m horizontal seams.

Does the order even matter? Yes it does in fact – Denote the set of n optimal vertical seams in an input image by S. If we start removing vertical seams first, the seams in S will be removed. However, if we first remove m horizontal seams, potentially many pixels within the seams in S have been removed, and this potentially affects the cost associated with the remaining pixels. This can now cause us to choose **different** vertical seams that are in an entirely different location in the image.

In your code you will implement the seam carving algorithm while supporting the 3 carving schemes we have just discussed :

Scheme 1 : Removing vertical seams first to resize the image's width, then removing horizontal seams from the resulting image.

Scheme 2: Removing horizontal seams first to resize the image's height, then removing vertical seams from the resulting image.

Scheme 3: Removing seams intermittently : vertical, horizontal, vertical, horizontal etc.

General implementation notes:
- Note that at least for seam duplication we need to remember the coordinates of seams we remove with respect to the original image. How can this be done?
- The costs matrix may contain very large values. You're advised to use matrix of **double** values.
- Use the forward formulation for pixel energy (described in recitation).
- As stated above, your code should support three different Seam Carving schemes for resizing an image : vertical seams first, horizontal seams first, and intermittent.
- Note that the carving scheme is **only** relevant when an image is resized in both dimensions.
- Your code **does not** need to support intermittent carving when scaling an image up in both dimensions - Do you see what would be the problem in trying to implement that? This is a good question to make sure you have a good general understanding of the algorithm.
- You first need to implement code that only supports downscaling, supporting upscaling is a bonus (see details below).
- We will check your work manually; your results should look similar to ours (not necessarily identical, but close enough).
- Your code should **not** run much slower than the supplied JAR.

PART B – Seam Carving : code

BasicSeamsCarver extends **ImageProcessor** – A class that **scales down** an image using Seam Carving. The rgbWeights and carvingScheme fields should affect the resulting image.

`Constructor(BufferedImage workingImage, int outWidth, int outHeight, RGBWeights rgbWeights)` – Initializes some necessary fields. Not fully implemented. You are required to continue its implementation; initialize some additional fields, you may run some preliminary calculations etc. Note that as mentioned, this class should **only** support downscaling. This means you can assume that outwidth, outheight are not greater than the width and height of workingImage.

`carveImage(CarvingScheme carvingScheme)` - Performs Seam Carving in order to reduce the height and width of the input image according to the dimensions provided in the constructor. The specific seam carving procedure used is determined by the carvingScheme parameter.

`showSeams(boolean showVerticalSeams, int seamColorRGB)` – Carves either vertical or horizontal seams from the input image (based on the showVerticalSeams parameter), and then colors the seams that were carved upon the input image.

AdvancedSeamsCarver extends **BasicSeamsCarver(implementing this class is a bonus)** – This class uses and expands upon the functionalities of BasicSeamsCarver to allow scaling an image up using Seam Carving. This m

`Constructor(BufferedImage workingImage, int outWidth, int outHeight, RGBWeights rgbWeights)` – Initializes some necessary fields. Not fully implemented. You are required to continue its implementation; initialize some additional fields, you may run some preliminary calculations etc. Note that as this class **needs** to support upscaling we **no longer** make the assumption on outWidth and outHeight we made in the previous class – each **may** be greater than the input image width/height.

`resizeWithSeamCarving(CarvingScheme carveScheme)` – Resizes the input image by using seamCarving. Once again, no simplifying assumptions are made on outWidth, outHeight. You need to support all types of resizing.

Bonus

These things can give you extra points, but you're on your own here. We won't answer questions regarding the bonus points.

1. (5pt) Implement the AdvancedSeamsCarver class.

You can submit a partial implementation for partial points – if your AdvancedSeamsCarver supports everything except for increasing an image's height you can get 3 points. For providing the remaining functionality you receive an additional 2 points.

Note that the implementation of bonus items must be documented in an accompanying `readme.txt` file. **Undocumented items won't be graded.**

Appendix I – Additional implementation tips

- A gray color g can be coded as: **new** Color(g, g, g).getRGB();

- Note that there is an inversion between the coordinate system of an image to the coordinate system of a matrix, this can be very confusing at first. If we represent an image with a 2D matrix M, pixel (x,y) is located in M[y][x] – think about why this is true : In a matrix the first coordinate is associated with height.

- When removing a vertical seam, all pixels to the right of it are shifted left by one step. When removing a horizontal seam, all pixels below it are shifted up by one step. You will have to remember the original position of pixels you remove along the carving process (particularly for showSeams and for upscaling). Use a helper array for that, e.g. an array A with the size of the image, where A[i][j] contains the **original** coordinate of the pixel located in (j,i) in the carved image we currently hold.

    For example :

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

    (note that the seam we remove is underlined), becomes :

(0,1)	(0,2)	(0,3)
(1,0)	(1,2)	(1,3)
(2,0)	(2,2)	(2,3)
(3,0)	(3,2)	(3,3)

- At the top row ($y = 0$), the cost would be:

    $$M_{0,x} = pixelEnergy(0, x).$$

- Define a corner pixel to be a pixel where x = 0 or x = width – 1 when dealing with vertical seams, and y = 0 or y = height – 1 when dealing with horizontal seams.
    In a corner pixel, we don't have all of the options for computing the cost. What you should do is use what you have, for example if x = 0, and we are dealing with a vertical seam :

    $$M_{y,0} = pixelEnergy(y, x) + \min \begin{cases} M_{y-1,0} + C_V(y, 0) \\ M_{y-1,1} + C_R(y, 0) \end{cases}$$

    In the official solution we also set C_V in this case to be 255 in order to discourage seams from travelling through corner pixels. You don't have to do this but if you can if you want to achieve results that are identical to the provided JAR.

- If you are having a hard time understanding the FunctionalForEachLoops class : the forEach method receives a method reference for some method f(y,x) and simply executes the following piece of code:

```
For (int y = 0 ; y < height ; y++){

    For (int x = 0 ; x < width ; x++){

        f(y,x)

    }

}
```

Once again you don't have to use this, you can just use standard for loops.

Good Luck!