Programming 2 (SS 2022)
Saarland University
Faculty MI
Compiler Design Lab

Project 2

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Marcel Ullrich, B. Sc.

Seam Carving in C (15 Points)

(a) Original (b) Scaled (c) Cropped (d) Seam Carved

In this project you will write an application that performs seam carving (content-aware image resizing). Seam carving is a method to scale the aspect ratio of images that aims to keep the proportions and shape of relevant contents and tries to reduce distortion. It was originally presented by Shai Avidan at Mitsubishi Electric Research Labs (MERL) and Ariel Shamir (Interdisciplinary Center and MERL) at the computer graphics conference SIGGRAPH 2007. An in-depth description of the algorithm, image and video examples as well as further use cases and extensions can be found on the website of the inventors:

`http://www.faculty.idc.ac.il/arik/SCWeb/imret/`.

Since this is your first C project, the structure of the program as well as input processing is already implemented. You only have to implement the appropriate functions in `image.c`, `energy.c` and `main.c`. Do not edit `argparser.c` and the `Makefile` as well as the `main` function in `main.c`, and the functions `image_init`, `image_destroy` and `image_read_from_file` in `image.c`.
You can compile and link the project by just executing `make` in the project's folder on the command line. The output files of the compiler – including the built application `carve_debug` (and `carve_opt` see section 5) – can be found in the `bin` folder. You can check out the project using

```
git clone https://prog2scm.cdl.uni-saarland.de/git/project2/<username> project2
```

Remember to replace `<username>` with your personal user name.

1 Seam Carving

The algorithm generates a dispensability score for all areas/pixels of an image. Using this score, paths along the most dispensable pixels are searched that lie crosswise to the compressed or elongated direction. At these seams, pixels are added or removed. Different variants of the base algorithm usually distinguish themselves by the strategies they use to calculate the dispensability score. In the base version, we are interested in, it is done by considering only the difference between the color values of neighboring pixels. We call this difference the *energy*. Pixels with the least energy are the most dispensable ones. We also treat only the downsizing case.
Essentially, the algorithm works as follows:

- Calculate the *local* energy of every pixel
- Calculate the energy of all adjacent vertical pixel-paths
- Find the pixel-paths with minimal energy
- Remove the pixels of this path from the image

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

(a) Pixels (in blue) required to compute the local energy of the pixel at position (1,1)

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

(b) Pixels (in blue) required to calculate the energy of the seam at (1,1)

Figure 2: Illustration of the pixels required for the energy calculation

1.1 Algorithm

Local Energy. The goal of the algorithm is to find a connected pixel-path (from now on called seam) with minimal energy and to remove it from the image. To calculate the energy of a seam, the energy of every pixel has to be calculated first. Every pixel has a *local* energy which is the sum of the computed color differences (see section 2.4) between the current pixel and its left neighbor (if present) as well as the upper neighbor (if present). See figure 2a.

Total Energy. Formally, a list of integers (representing x coordinates of the image) is called a *seam* if:

- the difference of subsequent elements is -1 (below left), 0 (straight below) or 1 (below right),
- for any element x we have $0 \leq x < w$, where w is the width of the image
- and the list has length h, where h is the height of the image.

This captures the intuition of a (vertically or diagonally) connected pixel-path from top to bottom. A prefix of a seam is called *partial seam* (i.e. a connected pixel-path from the top most row to some pixel inside the image).
The energy of a (partial) seam is the sum of the local energy values of all contained pixels. Since we are only interested in the seam with minimal energy, at every pixel p it suffices to compute the energy of the partial seam with *minimal* energy, that reaches p (there is a simple proof by contradiction for this statement). This value is called the *total energy at p*.
The total energy at a pixel p is calculated starting at the top by adding minimum of the *total* energy of the three pixels directly above (top left, top center, top right) to the local energy of p (see figure 2b). If any of the pixels is not present, it is excluded from the minimum calculation. If none of the three pixels exist, i.e. the pixel is in the top most row, nothing is added to the local energy.
The optimal to-be-removed seam is the path of connected pixels that have a minimal total energy. After calculating the total energy at every pixel, for any pixel the total energy is the accumulation of the local energies of the pixels above. To identify the optimal seam, the pixel in the bottom most row with the lowest total energy value has to be found. From this pixel, the optimal seam can be reconstructed.
To disambiguate which seam shall be removed, we use the following rule: If there are multiple optimal seams with distinct positions in the last row, the seam with the lowest x-coordinate is to be removed. If a pixel has multiple optimal neighbors, first the top center neighbor and then the top left neighbor is to be preferred.

1.2 Dynamic programming

The calculation of the energy and optimal seam would also be possible using a "brute-force" approach: Computing the accumulated cost of a pixel requires the costs of the neighbors contrary the seam's direction. A naive approach would be to calculate the cost by calling the same method recursively until the top row is reached. In doing so, the cost for many pixels would be calculated multiple times. This is due to two neighboring pixels having many common paths through which they are reachable. This disproportionately increases the computation time for larger images, therefore rendering the technique useless for the end user.

The variant described in 1.1 employs *dynamic programming* to solve this efficiency issue: The computation starts in the top row and stores the accumulated costs of every pixel into an array. In the next row, the results from earlier can be loaded from the array instead of recursively recalculating them.

2 General remarks for working on the project

2.1 Command line arguments

Your application uses the following command line syntax: [1]

```
carve_debug [-s] [-p] [-n <count>] <image file>
```

The following specifies the command line arguments individually:

- `-s` prints statistics from the input image and exits the application (see section 3.1).
- `-p` prints the column indices of the optimal seam and exits the application (see section 3.2).
- `-n <count>` executes `<count>` steps of the algorithm; if not specified, the full application is run (see section 3.3).
- `<image file>` the input image; note that this argument may be at any position in the command line and *not* necessarily at the end.

In case multiple command line arguments are specified, `-s` has the highest priority, followed by `-p` and finally `-n`. Only the functionality requested by the argument with the highest priority is executed. For example, if `-s` and `-p` are given, the statistics are printed and then the application is terminated.
If only the input image is given, the full algorithm runs (see assignment 3.3). If the input image is not given, the application exits with `EXIT_FAILURE` (see `man 3 exit`).
The argument parser is already implemented in `argparser.c` and the handling of the arguments is also already implemented in the `main` function of `main.c`.

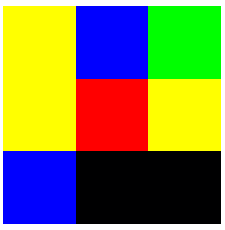
2.2 Image format

We use the *portable pixmap format* as in- and output format. The file ending for *RGB*-images in that format is `ppm`. Those files have the following content:

- The first line contains only `P3` followed by a `newline`
- The second line contains the width w and height h of the image as integers separated by at least a single whitespace. The line is ended by a `newline`
- The third line contains the number `255` followed by a `newline`.
- Afterwards $h * w$ pixels follow. Every pixel comprises three numbers that represent the red, green and blue values respectively. For every component, the minimal value is `0` and maximum is `255`.

```
P3
3 3
255
255 255   0   0   0 255   0 255   0
255 255   0 255   0   0 255 255   0
  0   0 255   0   0   1   0   0   1
```

(a) PPM Format



(b) Decoded image

Figure 3: Example PPM format

[1] Analogously there is an optimised binary `carve_opt`, see section 5 for more on these two variants.

Possible issues with the input image are too few/many pixels, width/height values smaller than or equal to zero, and more content-related problems.
The already implemented function `image_read_from_file` in `image.c` does the file reading for you. It reads an image from file and stores the result in a struct. It exits the application with the `EXIT_FAILURE` exit code if the input file is corrupt.

2.3 Representation of images in C

Images are represented as arrays of 8-bit integers. Three consecutive integers represent the respective colors of a pixel. For improved handling, we use a `struct pixel` that encapsulated the three 8-bit integer components. The image then is represented by an array of `struct pixel`s. To represent a two-dimensional image with the width w and height h, a single dimensional array of size $w \cdot h$ is required. The pixels are stored row-wise: the pixel with the coordinate x, y is at index $x + w \cdot y$ in the array, where w is the width of the image in pixels. You should use the function `yx_index` (defined in `indexing.c` for your index accesses.
You will also have to represent the total energy values at every pixel as a matrix such that you can use the same array index for the pixel and the corresponding total energy value.

2.4 Color differences between pixels

We define the sum of squares of the differences of the individual color components as the color differences:

$$(red(x) - red(y))^2 + (green(x) - green(y))^2 + (blue(x) - blue(y))^2$$

2.5 Additional remarks

- We urge you to keep the provided `Makefile` as-is, as we replace this file with our own version for the evaluation.
- Please do not commit additional image or other non-source files in the git repository.
- You are only allowed to edit: `main.c`, `image.c`, `energy.c` and `indexing.c`, as well as `image.h`, `energy.h` and `indexing.h`.
- Do not create additional source or header files.
- You may add additional functions and entries in the existing source code and header files, but don't change the existing signatures.
- Do not delete existing functions, not even in the case where you don't implement them.
- When starting the implementation, the test images `small1` and `small2` will be helpful (see section 4).
- Use `git`! As soon as you start implementing, you should always commit and push your changes, whenever you are changing something.

3 Assignments

3.1 Print statistics (1 points)

Implement the function `statistics` in `main.c` that prints statistics from the input image. When the application is executed with the command line parameter `-s` this function is called before the application terminates afterwards. The function should print the image width and height as well as its brightness. Compute the brightness as described in the following:

- We define the brightness of a pixel as the sum of the three color channels divided by three:

$$(red(x) + green(x) + blue(x))\ /3$$

- You obtain the brightness of the image by adding up the brightness of all pixels and dividing that by the number of pixels in the image.
- Use integer arithmetic at all times, do *not* use commercial rounding.

Use the following format for the output and do *not* write anything else to `stdout`:

```
printf("width: %u\n", <width>);
printf("height: %u\n", <height>);
printf("brightness: %u\n", <brightness>);
```

3.2 Output minimum energy path (7 points)

Implement `find_print_min_path` in `main.c` that prints the path with *minimal* energy as found by the seamcarving algorithm. It is called when the application is executed with the command line parameter `-p`.
To find the minimal path you have to implement the algorithm as it is described in 1.1. You will also find detailed information about the functions to implement and what they should do in the comments of the source code.
The printing has to respect the following specification:

- For every image row, print an `x`-index that is on the selected path.
- Write one index per line.
- Output the path from top to bottom.
- Do *not* write anything else to `stdout`.

3.3 Seam carving of the image (7 points)

Implement the function `find_and_carve_path` in `main.c` that executes `n` steps of the algorithm and writes the result to `out.ppm`.
It is called when the application is run with the command line parameter `-n <count>`.
Use the functions implemented in 3.2 and refer to 2.2 for the output format. Respect the following specification:

- Every step reduces the image's width by one pixel; in particular `-n 0` results in a copy of the input image.
- For every pixel the width of the image is reduced, add a black pixel to the right side of the image. This results in the output image having the same width as the input image, however the output image has a black border of width `<count>` on the right.
- You can expect the argument `n` to be greater or equal to zero and less or equal to the width of the image.
- Do not print anything to `stdout`.

4 Tests

You can run the *public* tests yourself from the base directory with the command:

```
./test/run-tests.py
```

To compile and run all tests you can use:

```
make check
```

The example input images are found in the `test/data` directory. The reference output for the tests is found in the `test/ref_output` directory. You have to pass all public tests of an assignment to be eligible for any points for that assignment. The public tests are additionally run on our test server in conjunction with the private *daily* tests. You will be informed about the results of those tests via e-mail. After the project's deadline, your submission will be evaluated according to additional *eval* tests. We will only consider the state on the `master` branch of your git repository at *24.05.2022, 23:59*.

4.1 Individual tests

You can also execute individual tests, which is especially useful when larger parts of the implementation are still missing.

- You can list the names of all tests with `./test/run-tests.py -l`.
- `./test/run-tests.py -f <name>` executes the test `<name>` only.

5 Running the project

You compile the project by typing the command `make` in the base directory of your project. This creates three binaries in the directory `bin`:

- `carve_debug`: You should use this binary to debug your project because it tells you about memory leaks and is not optimized. Optimizing compilers modify the code a lot (to improve performance), which can make it very hard to debug. For example an optimization may reorder statements of your code, which could lead to unexpected behaviour.
- `carve_opt`: You can use this optimized version to test your program on bigger images.
- `testrunnner`: This is used by the testing script (`testrun-tests.py`), you shouldn't use it directly.

An example call would be:

```
./bin/carve_debug -n10 test/data/owl.ppm
```

5.1 Optional: Installing and Using `nomacs`

Unfortunately, we forgot to ship an image viewer on the VM, so you have to install the image viewer `nomacs` (or any image viewer of your choice) by executing this command:

```
sudo pacman -S --noconfirm nomacs
```

You can now inspect the generated image `out.ppm` or the test files (e.g. `test/data/small2.ppm`):

```
nomacs output.ppm
```

If you want to inspect specific pixels you can press `ctrl+I` (or select Panels → Toolbars → Statusbar) to show the statusbar, which tells you about the coordinate and RGB values of the pixel at the position of your cursor.