Image Retargeting & Object Removal Via Seam Carving

1st Tímea Gyarmathy
Applied Computer Science
Vrije Universiteit Brussel
Brussels, Belgium
Timea.Gyarmathy@vub.be

2nd Seppe Lampe
Applied Computer Science
Vrije Universiteit Brussel
Brussels, Belgium
seppe.lampe@vub.be

3rd Akshaya Ganesh Nakshathri
Electrical Engineering
Vrije Universiteit Brussel
Brussels, Belgium
Akshaya.Ganesh.Nakshathri@vub.be

4th Tamas Janos Paulik
Electrical Engineering
Vrije Universiteit Brussel
Brussels, Belgium
Tamas.Janos.Paulik@vub.be

***Abstract*—Image resizing is a common task in digital media, which is most commonly achieved through cropping, stretching or shrinking a picture. In this project we present content-aware image resizing via seam carving, which we also apply for object removal and content amplification. We explore the performance of the algorithms on both static frames and videos and introduce a user-friendly, intuitive graphical user interface which gives easy access to all of the above features in one application. We present the theory and practice behind detecting and using seams, which are an 8-connected path of low-energy pixels and we analyze multiple energy functions for extracting them. We achieve state of the art performance in terms of quality, while our execution time might still be improved upon.**

***Index Terms*—image scaling, content-aware resizing, seam carving**

I. Introduction

Image resizing is a common computer processing task, which is essential for i.a., proper display of web-pages and cinematographic content. As the display size of images on the web has become dependent on screen resolution, there exists a practical need to resize images in real time. However, this resizing can be performed in a multitude of manners. The most naive implementations make use of uniform functions or cropping. However, these methods do not consider the contents of the image, preferably the parts with the most information in the image are retained, while low information pixels can be removed.

Seam carving offers content-aware resizing through the removal or addition of pixels through seams i.e., low-energy paths [1]. Seams carving applies an energy function to estimate the amount of information/energy each pixel contains. Subsequently, seams are formed by finding the lowest energy paths horizontally and vertically. These seams can then be removed to reduce the image or can be averaged with one of their neighbours to obtain a new seam for insertion, which increases the image in size.

In this work, written for the course *Image Processing* of Prof. Dr. Adrian Munteanu at the Vrije Universiteit Brussel, we reproduce the algorithms described by Avidan and Shamir [1] in Python. Furthermore, a graphical user interface (GUI) was created to easily demonstrate practical implementations of the created algorithms and the algorithms have been extended for usage on videos.

II. Algorithm Description

A. Energy Functions

At the core of the project are the energy functions, functions responsible for calculating the energy in each pixel of an image. Avidan and Shamir [1] evaluates four methods i.e., e_1 energy, entropy, segmentation and histogram of gradients (HoG). e_1 energy is the derivative in x and y direction (Eq. 1), while entropy is calculated as the e_1 energy plus the Shannon entropy of the 9x9 neighbourhood of each pixel. The Shannon entropy is given in Equation 2, where $P(x_i)$ represents the chance a sample having the value of x_i. Segmentation applies the e_1 energy on segments of the image instead of the whole image, Christoudias, Georgescu, and Meer [2] describes the segmentation method. Lastly, histogram of gradients divides the angles in 11x11 windows around each pixel into eight bins. The values added to each bin correspond to the gradient of that angle. In practice, a gradient is divided between two bins e.g., with bins of 0, 20, 40, 60, 80, 100, 120, 140 and 160 (180 loops back around to 0), an angle of 164° with gradient of 100 would supply $\frac{16}{20}$ * 100 to the 160 bin and $\frac{4}{20}$ * 100 to the 0 bin. After binning this window, the e_1 energy of that pixel is divided by maximum value of these bins to obtain the actual energy of that pixel (Eq. 3). The e_1 energy assigns high values to pixels which have high frequencies and the HoG part is high for pixels at edges since these have a distinct gradient. Thus, the e_1 divided by HoG assigns the lowest energies to pixels with low gradients around clear edges. Avidan and Shamir [1] evaluated these four algorithms and reports that e_1 and HoG offer the best results in practice. Therefore, we implemented both e_1 and HoG, and included entropy as well.

$$e_1(I) = |\frac{\partial}{\partial x}I| + |\frac{\partial}{\partial y}I| \tag{1}$$

$$H(X) = \sum_{i=1}^{n} P(x_i) * log(P(x_i)) \tag{2}$$

$$e_{\text{HoG}}(I) = \frac{e_1(I)}{max(HoG(I(x,y))} \tag{3}$$

The e_1 energy can be calculated in a number of ways, here, we chose for one of the most commonly used implementations

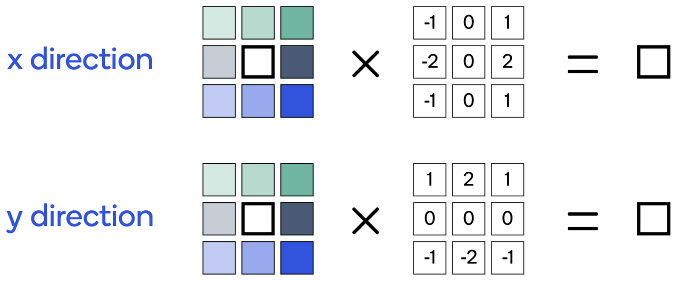


Fig. 1. The Sobel operator (www.developer.qualcomm.com).

i.e., the Sobel operator. This method multiplies 3x3 grids with the Sobel kernel to obtain the derivative in x or y direction of a pixel (Fig. 1). Note that this operation does not apply the common matrix dot product but rather the element-wise Hadamard/Schur product. The $sobel\_derivative$ function takes as input argument a 3x3 grid and returns the sum of the absolute values of the Sobel derivatives in x and y direction. Then $e1\_gray$ calculates the e_1 energy by applying the $generic\_filter$ function from the *scipy.ndimage* package. A $generic\_filter$ applies another function (here $sobel\_derivative$) in a sliding window approach to a multidimensional matrix. *e_1_colour* simply returns the average of applying $e1\_gray$ on the R, G and B band. Similar functions have been written for entropy i.e., $entropy\_helper$, which is the entropy equivalent of the $sobel\_derivative$ function, $entropy$, $entropy\_gray$ and $entropy\_colour$.

These functions are fully operational but run into performance issues when they need to be applied many times per second to large images. Therefore, we made numba-compatible versions of our code. Numba is a Just-In-Time (JIT) compiler for python code that specializes in improving the performance of python and numpy code [3]. As opposed to common 'pythonic' coding habits, numba prefers to have simple for loops instead of function calls or manipulations for the same operation. Additionally, numba does not support (yet) the use of other packages besides numpy. Moreover, numba has some other issues/complications making it difficult to i.a., operate on custom classes and pass functions as arguments. Lastly, one drawback of numba compilation is that function calls from within a numba compiled function can only be to other numba compiled functions. The energy functions have a numba compatible version, which are named properly with a 'numba' suffix e.g. $e1\_colour\_numba$. For the rest of the project functions have been made so that they are, when possible, compatible with numba. The HoG energy function only has a numba version as the regular implementation was impractically slow (>1 minute for the calculation of a 500*500 pixel image).

B. Seam detection

By applying an energy function to an image, an energy matrix is obtained. It is in this matrix that the optimal seams are searched. The iterative $find\_vertical\_seams$ function takes a matrix (energy matrix) as input parameter along with 'amount', the amount of vertical seams that should be returned. The function returns two arguments (i) a 2D array where each row represents a seam path, the values in the row contain the column indices of that seam path and (ii) the total energy of the last seam path. The function starts by iterating from the second row of the error matrix to the last. At each value (pixel) in each row the three neighbours above it are considered. The one which can be reached with the total minimum energy path (stored in the $path\_e$ variable) is considered its connecting neighbour, its index will be stored in the $path$ variable (not to be confused with $path\_e$). The $path\_e$ value of this connecting neighbour is added to the energy matrix value of the current pixel to obtain the minimum energy required to reach this pixel and will be stored in $path\_e$. After this loop the minimum value in the last row of the $path\_e$ matrix can be found. This represents the total energy of the minimum energy path (seam). If this value is equal to *np.inf* then no valid seam can be formed anymore. Since, one pixel cannot be included in more than one seam and seams do not run parallel, it is possible to run out of valid seam paths. In this case, the algorithm will stop looking for seams and returns the ones it has found. Note that this implies that the *amount* of seams requested to the algorithm is not necessarily the amount of seams it returns. The effective amount of seams that the algorithm could find can easily be obtained by applying a *len* function over the first (0^{th}) item the function returns. However, in the regular case that the minimum value in the last row is not *np.inf*, then the index of this value can be used in the last row of the $path$ matrix to backtrack the indices of this seam. $find\_horizontal\_seams$ can be used to find the horizontal seams, these are found by transposing the image and calling $find\_horizontal\_seams$.

C. Image reduction

Removing columns from an image is performed by the $remove\_column$ function. This function take an image and energy matrix as input and returns the image with one less column, the removed values and the indices (seam) of the removed values. $remove\_row$ transposes its input image and calls the $remove\_column$ function. Unfortunately, the $remove\_column$ function cannot be made numba compatible, as numba only partially supports numpy. Therefore, the numba-compilable $remove\_column\_seam\_numba$ has been created. As $remove\_column$ is more numpythonic (and easier to read) it has been left in the code but is not called in practice. $remove\_column\_seam\_numba$ takes as input an image and a seam and returns the image without that column. Again, the similarly named $remove\_row\_seam\_numba$ function first transposes the image and then calls $remove\_column\_seam\_numba$. These two functions return the new image and the values that were removed. The $remove\_rows_and\_cols$ function takes four parameters as input i.e., image, energy function, rows to remove and columns to remove and is not numba-compilable. As mentioned earlier, making functions, which take another function as argument numba compatible is difficult. Furthermore, making this function compilable for numba would not increase performance significantly as it just calls the predefined $remove\_column\_seam\_numba$ and $remove\_row\_seam\_numba$, which are already numba-compiled. As indicated by Avidan and Shamir [1], the rows and columns are removed one by one and the decision whether to remove a row or column is based on the energy of the horizontal and vertical seam with the lowest of the two being removed. $remove\_rows\_and\_cols$ returns four items i.e., (i) the new image, (ii) a list where each value corresponds to an array containing the removed values of one seam, (iii) a list with the removed seam indices and (iv) the order in which the rows and columns were removed, this is a list containing booleans, where *True* indicates a row and *False* a column. Note that this boolean list is only useful in theory when a square image is supplied or when a square image is obtained at a certain point during this process. When the image is not square, the length of the seam can indicate whether the seam and values belonged to a row or column. However, as there as states where a square matrix is obtained, this boolean matrix is required.

D. Image enlarging

Similar to $remove\_column\_seam\_numba$, an $add\_column\_seam\_numba$ function has been created. This function takes as input an image and a list of seams to add. This is in contrast to the $remove\_column\_seam\_numba$, which only takes one seam as input. This is the result of the fact that insertion and removal are, in practice, quite different. If one would insert e.g., 50 columns one by one, then the insertion would happen at the same place leading to local stretching artefacts. This is because the insertion will happen at the lowest seam, this new image will still have the lowest seam at that same area, causing the next insertion to happen at the same place. Removal on the other hand, removes the seam and thus, after removing a seam, it cannot be selected by the $find\_vertical\_seam$ function as it is no longer in the image. Therefore, we can remove seams one by one, without issue for optimal results, but addition should happen in bulk. This is why $add\_column\_seam\_numba$ takes a list of seams as input, instead of a single seam. Again, for the insertion of rows $add\_row\_seam\_numba$ transposes the input and calls $add\_column\_seam\_numba$.

The $add\_columns$ function takes as input an image, energy function and an integer indicating how many columns should be added. Before adding *n* columns, *n* seams should be found. This is why $find\_vertical\_seam$ has an optional parameter indicating how many seams should be returned. However, one cannot simply call $find\_vertical\_seam$ with the desired amount of columns to add. For example, if one wants to add 200 columns to a 150 by 150 pixel image, then it is impossible to find 200 columns in one step. Furthermore, adding 150 columns would just perform a content-unaware resizing. Thus, we should add X% of the column length, with X being a balance between preventing stretching artefacts and keeping the resizing content-aware. Note that the content of the image can place a limit on X and, as explained in Chapter II-B, that calling $find\_vertical\_seam$ with an *amount* does not necessarily return *amount* of seams. Thus, the amount of columns added at a time is dependent on the desired X% and the seam formation of the image. Here, we have set X to equal 50%. The add_rows again transposes the image and calls $add\_columns$.

Lastly, it should also be possible to re-introduce removed seams back into an image. Therefore, the $reconstruct\_column\_seam\_numba$ function takes as input an image, a seam and the removed values. This function re-introduces the values at the seam location in the image and returns this reconstructed image. $reconstruct\_row\_seam\_numba$ is the row equivalent of $reconstruct\_column\_seam\_numba$ and again transposes the image and calls $reconstruct\_column\_seam\_numba$ to perform the reconstruction.

E. Content amplification

We can use seam carving to amplify the content of an image while preserving its original size. This can be achieved by enlarging the original image by a desired amount and then reducing the image back to original size using seam carving.

The $amplify\_content$ method in the *SeamImage* class first enlarges the image by the specified $Amplification\_factor$ in the GUI, using OpenCV resize function. In the next step, our *resize* method is called with desired height and width of the image (which is same as the original size of the image) and this method in turn calls $remove\_rows\_and\_colums$ function, which then removes the desired number of seams vertically and horizontally to bring the image back to the original size with amplified content.

F. Seam Carving in the gradient domain

There are times when removing seams using seam carving generates visual artifacts in the output image. These artifacts can be minimized by performing seam carving in the gradient domain. The energy function is computed as described in II-A and the seams to be removed are also computed as described in II-B. These seams are removed from the original image and also the x and y derivatives of the original image (derivatives are computed as a simple forward derivative). A Poisson solver as described in [4] is used to reconstruct the image from the seam carved gradients.

A new function $process\_gradient\_domain$ was implemented which first computes x and y derivatives of input image, performs seam carving on input, x and y derivative images. Then, the $poisson\_solver$ is used to individually reconstruct the three color channels.

The $poisson\_solver$ function was implemented by referring to the existing python implementation in [5], which was based on the MATLAB implementation in [6]. Our own version of DST and IDST was written, which outputs results exactly same as the MATLAB code. The $poisson\_solver$ is computationally intensive to run. Hence it is updated to work with numba, in order to speed up the execution.

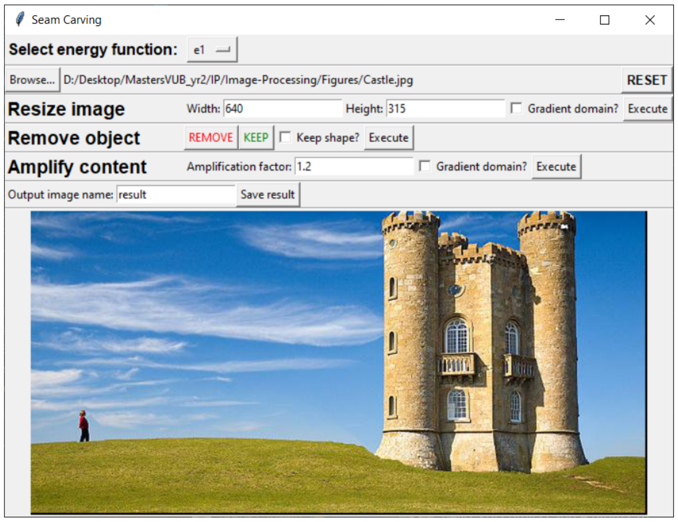
G. Object removal

For the object removal we created the $remove\_mask$ function, which takes the image, the energy function that we want to use, the masked area that we want to remove and that we want to keep, as an input. First, the determines, based on the smallest dimension of the mask to remove, whether it will call the $remove\_column\_seam\_numba$ or $remove_{r}ow_{s}eam_{n}umba$ function. Then, it calculates a mask by multiplying the mask to remove with a negative number relative to the size of the image and the mask to keep with a positive number relative to the image size. Then we loop until the mask is empty, before each seam removal we modify the values of the energy matrix with the mask. This results in small values (negative) for the pixels we want to remove and large values for those to keep, which will draw the seam carving algorithm towards the area to be removed and away from that to be kept. Furthermore, it is possible that, when the area to be removed and kept are close to each other that some pixels that require removal become enclosed by those that should not be removed. In this case, the algorithm will continue to remove seams until only those containing the mask to be kept are left, only then it will start selecting seams containing the last few pixels to be removed. To prevent this, and thus prevent losing the majority of our image, we have built in a check, which stops the loop when the ideal seam no longer contains pixels from the mask to be removed. Finally the function gives back the new image. We can add seams to the picture in order to keep the aspect ratio of the original image. To guarantee optimal functioning of this algorithm, no two spatially incoherent zones should be marked to be removed at the same time. One can indicate as many different patches of areas to be kept, but as the choice of vertical/horizontal seam carving is based on the difference between min and max column and row indices of the mask to be removed, marking multiple areas can lead to non-optimal behaviour. In short, if you want to remove two items, perform it in two operations.

H. Graphical User Interface

We created an intuitive, user friendly application which incorporates the functionalities listed in the above chapters. It gives direct access to execute the functionalities from chapters II-C, II-D, II-G, II-E, II-E and II-F, additionally it enables choosing between the functions detailed in chapter II-A. The GUI of the application is illustrated on Figure 2. It was implemented with the default Python GUI library *tkinter*.

The application keeps track of an implemented inner structure, a *SeamImage* class, which was implemented in order to easily manipulate images. Objects of this class can be instantiated by passing the location of an image to the constructor. This image is then extracted and stored as numpy array. This class keeps track of which seams and values were removed and added and in which order. The class also has methods for adding and removing rows and columns, which call the numba-compiled functions explained in Chapters II-C and II-D.



D:/Desktop/MastersVUB_yr2/IP/Image-Processing/Figures/Castle.jpg
Width: 640 Height: 315
Amplification factor: 1.2
Output image name: result

Fig. 2. The GUI of the application with the file *Castle.png* selected. At the very top of the window there's a dropdown to select between 3 energy functions: *e1*, *entropy* and *hog*. The path to the file is displayed under this, next to the *Browse...* button, and the three main seam carving applications are listed below that. Lastly, there is a save result functionality. The area where the image is displayed is interactive, the user can draw on it with Remove or Keep for the object removal functionality.

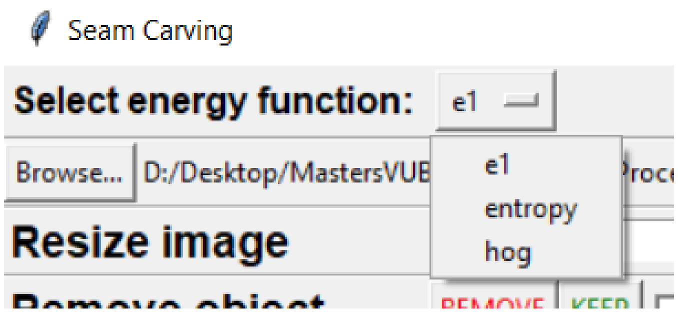
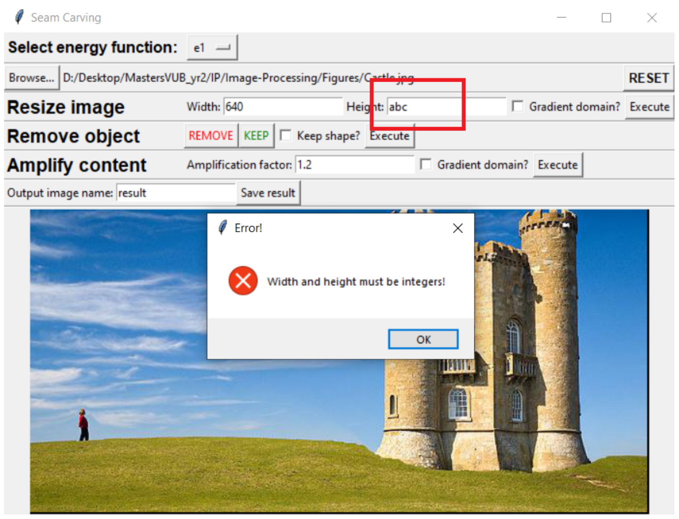


Fig. 3. The energy function dropdown selections. Each option in particular is described in chapter II-A.

We can identify two main areas of the GUI: the top with interactive components and the bottom with the image display, and additional drawing capabilities for the *object removal* functionality. The top part has six main functional sections, separated logically by horizontal divider lines. The first one is an overall dropdown for selecting the energy function to be applied in any seam carving function. The second and the last sections handle image input and output, while the middle three provide the components for seam carving functionalities. In the following subsections the functionality of the GUI, depicted in Figure 2, is discussed from top to bottom.

1) Energy functions: Energy functions for seam carving as described in chapter II-A can be selected from the top dropdown, next to the label **Select energy function**. The selections are ”e1”, ”entropy” and ”hog” as illustrated on Figure 3. The current selection will be applied to any seam carving function executed. Note that for time performance, e1 is superior to entropy and hog.


Seam Carving

Select energy function: e1

Browse... D:/Desktop/MastersVUB_yr2/IP/Image-Processing/Figures/Castle.jpg RESET

Resize image Width: 640 Height: abc ☐ Gradient domain? Execute

Remove object REMOVE KEEP ☐ Keep shape? Execute

Amplify content Amplification factor: 1.2 ☐ Gradient domain? Execute

Output image name: result Save result

Error!

Width and height must be integers!

OK

Fig. 4. The error popup when incorrect values are specified in entry boxes. Notice in the red bracket the input was 'abc' which, when clicking the *Execute* button triggered the error message "Width and height must be integers!"

2) *Opening files:* The *Browse...* button on the top of the application opens a file explorer where the user can select an image or video input to be displayed on the screen, in the bottom frame. At startup the application shows a black area where the image should be displayed or in case of video, the first frame. After selecting a file, the absolute path to the selected file is displayed next to the *Browse...* button. In the file explorer only supported files are displayed for selection with `.png`, `.jpg`, `.jpeg` or `.avi`, `.mpg`, `.mp4` extensions and only one file can be selected at once. If the user successfully selects an supported file, the bottom area gets resized to the shape of the image to be able to fully fit it into the frame.

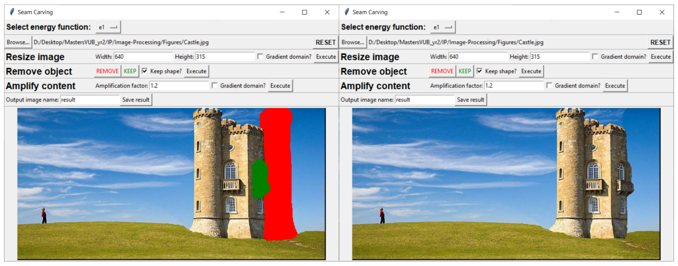
In the top right corner there is a *Reset* button which discards any drawings on the image or modifications done to the image via seam carving and displays the original image file while also resetting the internal *SeamImage* structure.

3) *Resizing images:* The components corresponding to resizing an image via seam carving described in sections II-C and II-D are located next to the label **Resize image**. There is an entry for *width* and one for *height*. The entries display the current width and height of the image at all times. The user can input a desired height and/or width and then call the seam carving procedure via clicking the button *Execute* to the right of the entry boxes. Inputting a non-integer value or a negative integer results in an error popup as illustrated on Figure 4.

Additionally, a checkbox is available for choosing if the operation should be executed in the gradient domain.

If a successful resize was called, the resized image will be displayed in the bottom area and the height and width of the area will be adjusted accordingly.

4) *Remove object from image:* Next to the label **Remove object**, we have two buttons, labeled REMOVE and KEEP. The default drawing color of the user is red, and drawing on the image can be executed anytime and would look like the illustration on Figure 5 on the left. Besides the button *Execute*


Seam Carving

Select energy function: e1

Browse... D:/Desktop/MastersVUB_yr2/IP/Image-Processing/Figures/Castle.jpg RESET

Resize image Width: 640 Height: 315 ☐ Gradient domain? Execute

Remove object REMOVE KEEP ☑ Keep shape? Execute

Amplify content Amplification factor: 1.2 ☐ Gradient domain? Execute

Output image name: result Save result

Fig. 5. The drawing function for the object removal via seam carving on the left. One of the towers of the castle was marked for removal while the windows were marked as areas to be kept. On the right we have the result after pressing *Execute*. Notice that the checkbox for "Keep shape?" was ticked, which yielded a result image with the same width and height, as confirmed by the numeric fields displayed on the GUI.

for calling the seam carving functionality described in section II-G, we have a checkbox here labeled *Keep shape?*. Ticking this checkbox will maintain the original width and height of the image after removing an object.

5) *Amplify content:* The functionality described in section II-E can be called via the button execute inline with the label citing **Amplify content**. An amplification factor can be specified which by default is set to 1.2. Logic for validating the input is added at this entry as well, as described before at subsection II-H3.

Additionally, a checkbox is available for choosing if the operation should be executed in the gradient domain.

6) *Save resulted image:* The last functional section above the image display provides a function to save the result as PNG if it was an image or an MP4 if a video was input, by clicking the button *Save result*. Next to the label "Output image name:", the user can specify a file name. By default, this is set to `result`, which would create a file named `result.png` or `result.mp4` if saved. The output files are placed into the same folder where the application is ran from.

I. *Extending seam carving to video*

We extended seam carving to video using the same settings (chosen in GUI) for every frame of the video. All features mentioned above, namely Image resizing, Remove object and content amplification are supported for videos too. In the case of video, each frame is read individually, a new SeamImage object is created for that frame and the required function is processed. At the end of the process, the processed frame is written onto a temporary video file. Once all the frames are processed in a loop, the temporary video file gets fully updated. If the user chooses to save the result, then this temporary video gets renamed to the chosen file name in the GUI. In this implementation every frame is treated as a different image, when in-fact there is lot of inter-frame similarities, which can be taken into account to optimize implementation but is out of scope for this project.

III. RESULTS

Figures 6 and 7 show the results of content aware image enlarging, both pictures are widened successfully. Figures 8, 9, and 10, are examples of the object removal function, where we can set both the area to be removed and to be kept. For figure 8, the solution is elegant and indistinguishable. Figure 11 represents the amplification function of our result. Figure 12 has the input image (left), resized image with seam carving (middle) and resized image with gradient domain seam carving (right). We can observe that resizing in gradient domain provides a smoother blending of pixels e.g., on the bottom right of the images. For seam carving on video, the flag pole was removed from the input video using object removal in GUI. Both the videos with and without *keep shape* have been generated.

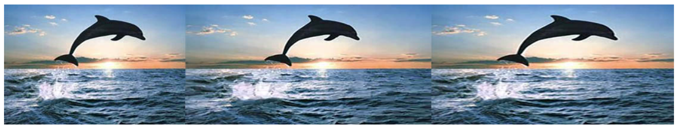


Fig. 6. On the left is the original image, which is in the middle widened via seam carving and without seam carving on the right.

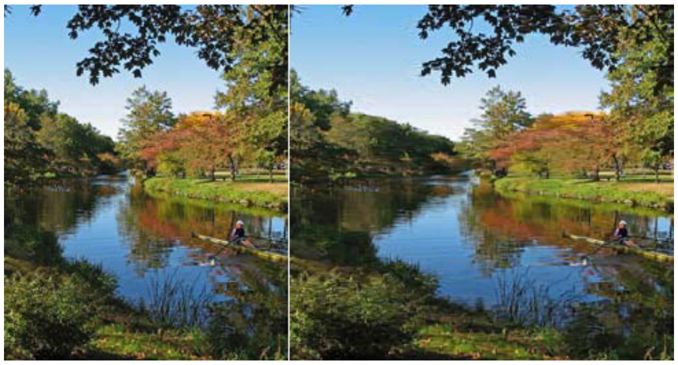


Fig. 7. The picture on the right is modified with seam carving.

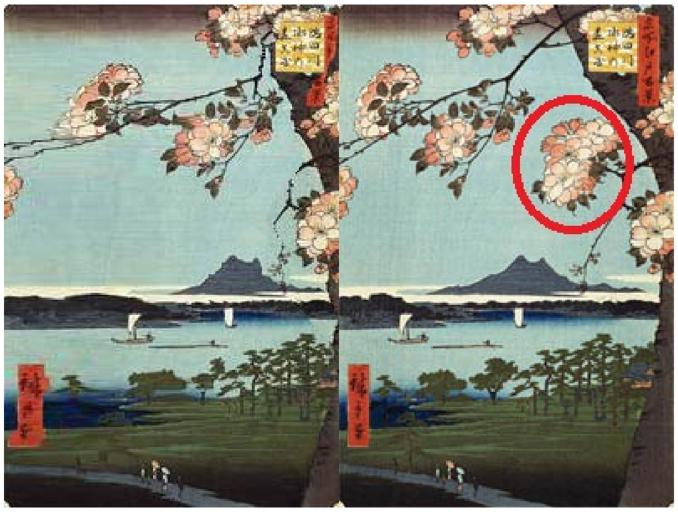
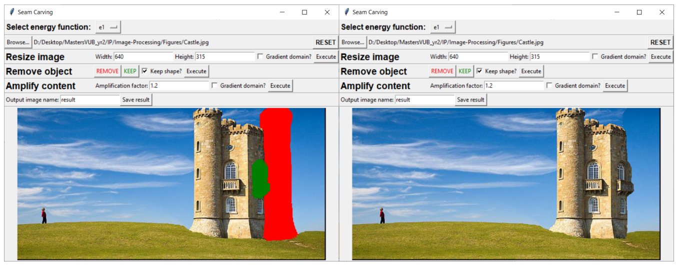


Fig. 8. Object removed by seam carving technique.



Select energy function: e1
D:/Desktop/MastersVUB_yr2/IP/Image-Processing/Figures/Castle.jpg
Width: 640 Height: 315 ☐ Gradient domain?
☑ Keep shape?
Amplification factor: 1.2 ☐ Gradient domain?
Output image name: result

Select energy function: e1
D:/Desktop/MastersVUB_yr2/IP/Image-Processing/Figures/Castle.jpg
Width: 640 Height: 315 ☐ Gradient domain?
☑ Keep shape?
Amplification factor: 1.2 ☐ Gradient domain?
Output image name: result

Fig. 9. Object removed by seam carving technique.

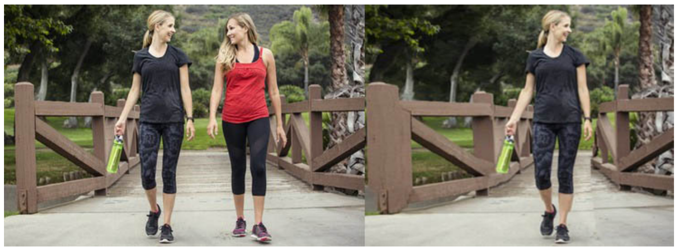


Fig. 10. Object removed by seam carving technique.

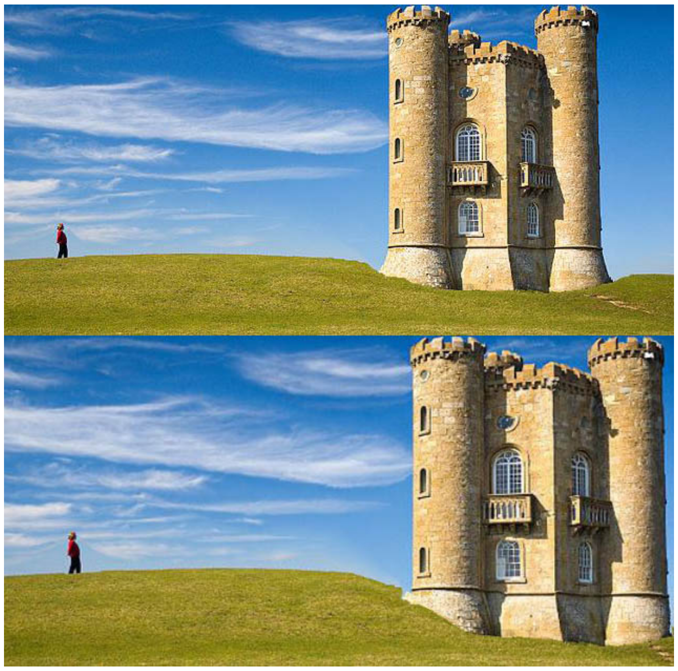


Fig. 11. Object amplification.

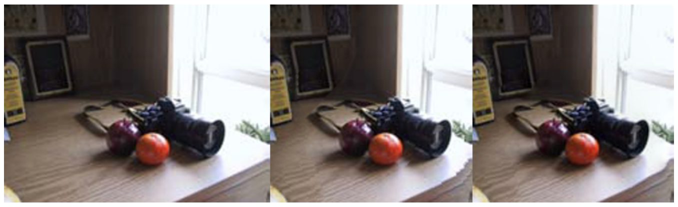


Fig. 12. Image resizing in gradient domain

IV. Discussion & Conclusions

Here, we presented a user-friendly application for demonstrating the significance of a number of seam carving applications. The algorithms are largely based on the findings by Avidan and Shamir [1] and include aspect ratio change, image re-targeting, content amplification, object removal and gradient domain seam carving. For easy usage we created a Graphical User Interface and improved our method to handle videos as well. The algorithms have state-of-the-art performance in terms of quality, despite significant efforts e.g., numba compilation, state-of-the-art in terms of time performance i.e., real-time execution, could not be achieved. Modern-day techniques attempt to emulate seam carving by applying deep learning architectures, which are, after training, significantly faster to evaluate than the actual algorithms [7]–[10]. However, given the amount and size of the computations involved in our algorithms, real-time execution (without deep learning emulators) would require significant computing power and/or many low-level optimizations, and was thus out of the scope for this project.

References

[1] S. Avidan and A. Shamir, "Seam carving for content-aware image resizing," in *ACM SIGGRAPH 2007 papers*, 2007, 10–es.

[2] C. M. Christoudias, B. Georgescu, and P. Meer, "Synergism in low level vision," in *Object recognition supported by user interaction for service robots*, IEEE, vol. 4, 2002, pp. 150–155.

[3] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[4] P. Perez, M. Gangnet, and A. Blake, "Poisson image editing," in *ACM Trans Graph*, 2003, pp. 313–318.

[5] J. Doerner. (). "Fast poisson reconstruction in python," [Online]. Available: https://gist.github.com/jackdoerner/b9b5e62a4c3893c76e4c.

[6] R. Raskar. (). "Matlab code for poisson image reconstruction from image gradients," [Online]. Available: http://web.media.mit.edu/~raskar/photo/code.pdf.

[7] J. Ye, Y. Shi, G. Xu, and Y.-Q. Shi, "A convolutional neural network based seam carving detection scheme for uncompressed digital images," in *International Workshop on Digital Watermarking*, Springer, 2018, pp. 3–13.

[8] L. F. S. Cieslak, K. A. Da Costa, and J. PauloPapa, "Seam carving detection using convolutional neural networks," in *2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Ieee, 2018, pp. 000 195–000 200.

[9] E. Song, M. Lee, and S. Lee, "Carvingnet: Content-guided seam carving using deep convolution neural network," *IEEE Access*, vol. 7, pp. 284–292, 2018.

[10] L. Nataraj, C. Gudavalli, T. M. Mohammed, S. Chandrasekaran, and B. Manjunath, "Seam carving detection and localization using two-stage deep neural networks," in *Machine Learning, Deep Learning and Computational Intelligence for Wireless Communication*, Springer, 2021, pp. 381–394.

Appendix

Tímea Gyarmathy fully implemented the GUI as described in Chapter II-H and took care of the its integration with backend logic, and added any additional functionalities needed for better interaction.

Seppe Lampe took care of implementing the energy functions (Chapter II-A), seam detection (Chapter II-B), image reduction (Chapter II-C) and image enlarging (II-D). Furthermore, did he implement the numba compilation, create the *SeamImage* class used for the backend-GUI integration explained in Chapter II-H and aided in integrating the object removal (Chapter II-G) with the rest of the project.

Akshaya Ganesh Nakshathri took care of implementing Content Amplification (Chapter II-E), Seam Carving in the gradient domain (Chapter II-F) and extending seam carving to video (Chapter II-I).

Tamas Janos Paulik implemented the object removal part.