Image Inpainting and Selective Motion Blur

Gaurav Verma
Dept. of Electrical Engineering, IIT Kanpur
14244, gverma@iitk.ac.in

Abstract: This report is presented as a part of the coursework for EE604A, Image Processing (Fall Semester, 2017-18) at Indian Institute of Technology Kanpur. The report summarizes the implementation of two popular image processing techniques: (1) *Image inpainting* and (2) *Selective motion blur*. The MATLAB® code has been made publicly available [here]. The implementation of the inpainting technique is based on the work done by Criminisi et al. in 2004 [1]. Implementation of selective motion blur is based on the work of several researchers.

1. Image Inpainting

1.1. The Problem Statement

Image inpainting has been conventionally defined as the process of reconstructing lost or detoriated parts of images. The entire workflow involves an input image – an image which has some *defect* to it, and an output image that is free of that *defect*. Usually the defect is of such nature that it leads to absence of some part of the original scene(which was captured) in the current version. The output is an image that reconstructs the part that was lost. The following figure (Fig. 1) shows an example of a defective image and its restored version.

Fig. 1. Image Restoration. An example where the defective image (in which some part of the captured scene went missing) was restored using inpainting. Image Credits: R Walters from USA.

However, this isn't where inpainting ends. Restoration is just an application of it, just as object removal is. Inpainting is also used to remove large objects from the image, as shown in the following figure (Fig. 2). Here the challenge is to fill in the hole that is left behind in a visually plausible way.

1.2. The Proposed Solution: An Intuitive Explanation

The algorithm proposed by Criminisi et al. effectively hallucinates new colour values for the target region in a way that looks "reasonable" to the human eye. The user selects a target region, Ω, to be removed and filled. The source region, Φ, is defined as the entire image minus the target region, $\Phi = I - \Omega$. There is a template window, Ψ, the size which is to be fine tuned depending on the textures in the input image. Figure 3 below, clarifies the notation that is being used here. For satisfactory results, the template window size must be set slightly larger than the largest distinguishable texture element, in the source region.

Throughout the process, each pixel of the image maintains a *color* value (or "null" if the pixel is unfilled) and a *confidence* value, which reflects the confidence in the pixel value. During the course of the algorithm, the patches along the fill front are also given a priority value, which determines the order in which they are filled. Then, the algorithm iterates the following three steps until all the pixels have been filled:

- **Computing patch priorities:** The proposed algorithm is a best-first filling algorithm that depends entirely on the priority values that are assigned to each patch on the fill front. The priority computation is biased toward

Fig. 2. Removing large object from images. Right: Original Image, Left: The region corresponding to the foreground person has been manually selected and then automatically removed. Image Credits: Criminisi et al. [1]

those patches with are the continuation of strong edges and which are surrounded by high-confidence pixels. This has been discussed in detail in the paper [1]. Henceforth, the patch on the fill front with highest priority will be referred to as, $\psi_{\hat{p}}$.

- **Propagating texture and structure information:** The patch with the highest priority, $\psi_{\hat{p}}$, is filled with the data extracted from the source region, Φ. Instead propagating texture using diffusion, which inevitably leads to image smoothing, the image texture is propagated by direct sampling of the source region. The patch in the source region that is most similar to the target patch is copied as it is. This suffices to achieve the propagation of both structure and texture information from the source Φ to the target region Ω. Henceforth, the patch in the source region that is most similar to the target patch is referred to as $\Psi_{\hat{q}}$. Valid patches $\Psi_{\hat{q}}$ must be entirely contained in Φ.
- **Updating confidence values:** After the patch $\Psi_{\hat{p}}$ has been filled with new pixel values, the confidence is updated in the area delimited by $\Psi_{\hat{p}}$ as : $C(q) = C(\hat{p}) \ \forall \ q \in \Psi_{\hat{p}} \cap \Omega$. This simple update rule allows us to measure the relative confidence of patches on the fill front, without image-specific parameters. As filling proceeds, confidence values decay, indicating that we are less sure of the colour values of pixels near the centre of the target region.

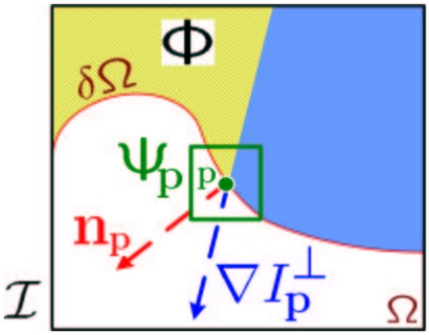

Fig. 3. Notation diagram. Given the patch Ψ_p, n_p is the normal to the contour $\delta\Omega$ of the target region Ω and $\nabla I_p^{\perp}$ is the isophote (direction and intensity) at point $\mathbf{p}$. The entire image is denoted with I. Image Credits: Criminisi et al. [1]

Extract the manually selected initial front $\delta\Omega^0$;
while $\Omega^t \neq \emptyset$ **do**
  1a. Identify the fill front $\delta\Omega^t$;
  1b. Compute the priorities $P(p)\ \forall p \in \delta\Omega^t$;
  2a. Find the patch $\Psi_{\hat{p}}$ with maximum priority, $\Psi_{\hat{p}}|\hat{p} = \text{argmax}_{\ p\in\delta\Omega^t} P(p)$;
  2b. Find the exemplar $\Psi_{\hat{q}} \in \Phi$ that minimizes $d(\Psi_{\hat{p}}, \Psi_{\hat{q}})$;
  2c. Copy image data from $\Psi_{\hat{q}}$ to $\Psi_{\hat{p}}$;
  3. Update $C(p)\ \forall p|p \in \Psi_{\hat{p}\cap\Omega}$;
end

Algorithm 1: Region filling algorithm

1.3. Examples from the Implementation

The following figures are the output of the MATLAB® implementation which is made publicly available [here]. The size of the template window, Ψ has to be varied, based on the input image. The size of the template window must be slightly larger than the largest distinguishable texture element, in the source region.

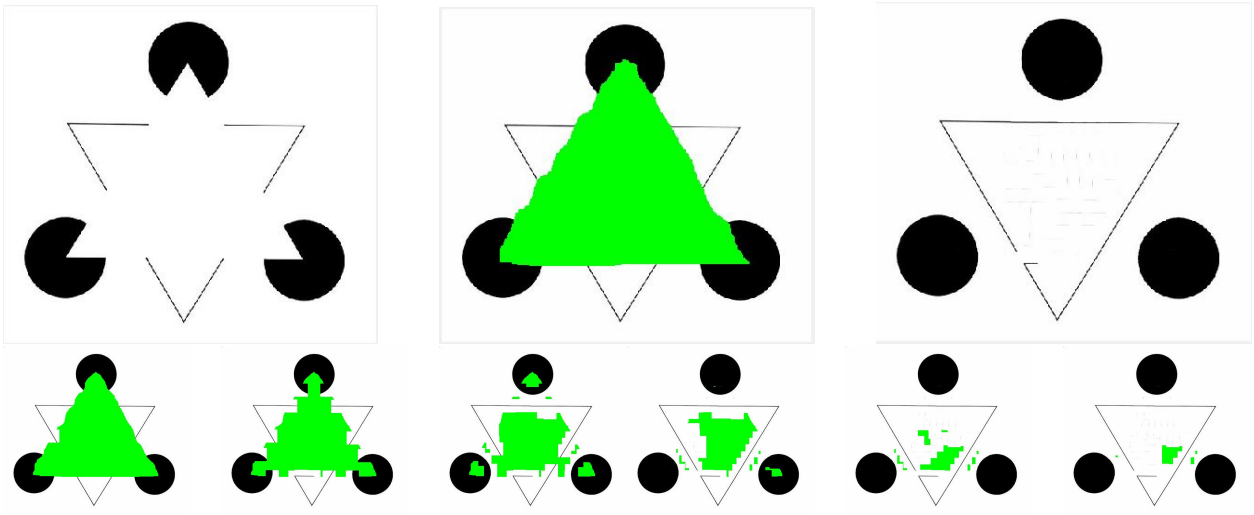

Fig. 4. **Examples from the Implementation**: **Row 1**: The boy in the foreground has been inpainted and the background has been reconstructed (psz = 15). **Row 2**: The jumping guy has been removed and the family photograph has been reconstructed. **Row 3**: The cow in the background has been removed (psz = 13). **Row 4**: The Kanizsa triangle was masked and the three circles and the triangle were perfectly constructed (psz = 35). **Row 5**: Depiction of region filling, as carried out by the algorithm. The patches along the edges are filled first, as discussed in the earlier sections.

As it can be noted, with the help of the example images provided above, the implementation works at par with the original implementation provided by Criminisi et al. [1]. The images provided in the fourth row of Figure 4, is the best example that can be used to understand the working of the algorithm intuitively. The following row (i.e., Row 5 in Figure 4) illustrates that the patches at the fill front, which lie along some kind of edge are filled first. This is what we intend to imply when we say that the algorithm is a best-first filling algorithm. It can also be noted, from images provided in the 2nd row of Figure 4, that along with removing objects, the algorithm also reconstructs the missing part of the image (as it reconstructed the semi-masked coin).

2. Selective Motion Blur

2.1. The Problem Statement

Selective motion blur deals with blurring everything, except the user-selected part of the image. The problem can further broken down into following two parts: (1) Segment out an object in the foreground that has been clicked and selected by the user. (2) The output image is a overlay of the two segments of the image, in which the user-selected segment remains as it is – crisp and sharp, and the other region has been burred out. The following figure illustrates this with the help of an example output image.

Fig. 5. **Selective Motion Blur**. The man in the foreground is sharp and crisp, while the background has been blurred out (in this case, it's a motion blur.) Image Credits: www.smashingmagazine.com

In the following sections, we have discussed the methodology that we have used to implement selective motion blurring. Few examples from the implementation have been also provided, along with their brief analysis.

2.2. *Methodology*

As mentioned, the entire problem has been broken down into two parts:
1. Segmenting the user-selected region out from the image
2. Overlaying the user-selected region and the blurred-version of the original image.
To deal with the first part, I have used lazy snapping. The Algorithm was given by Li et al. in 2004 [2]. Below, I have given a brief intuitive explanation of the algorithm.

Lazy Snapping:

The goal of the algorithm is to separate the foreground from the background. The first step involves selecting a few pixels from the foreground and the background. Contrary to the GraphCut algorithm, which aims at solving the labelling problem on a pixel level, Lazy Snapping aims to solve the the labelling problem on a segment level. The input image is presegmented with the Watershed Algorithm. This step is followed by GraphCut on the segments instead of pixel level which leads to a reduced complexity and hence the results are available at interactive speeds.

GraphCut Algorithm borrows tools from Graph Theory to partition an input image into foreground and background. The following illustration (Figure 6) has been borrowed from Prof. Guillermo Sapiro's Coursera lecture. Each pixel of the image is treated as a node, and two additional nodes are created (1) *Sink*, representing the background and (2) *Source*, representing the foreground. Every node representing a pixel is connected to the source and the sink with an edge of some weight. The weights of these edges correspond to the probability of that node being classified into the foreground or into the background. Next, the nodes representing the pixels are also connected among themselves with edges of some weight. For most of the cases, the connection is deemed established with the 4-connected neighbours (or 8-connected neighbours) of a pixel. The weights of these inter-pixel edges have to be something that promotes similar pixels to stay together and different pixels to stay apart – similarity being defined as per the application. Once the entire graph has been constructed, we partition it using Min Cut algorithm. The Min Cut algorithm partitions the graph with minimal effort – along the edges that are weak. This partition divides the image into a foreground and a background.

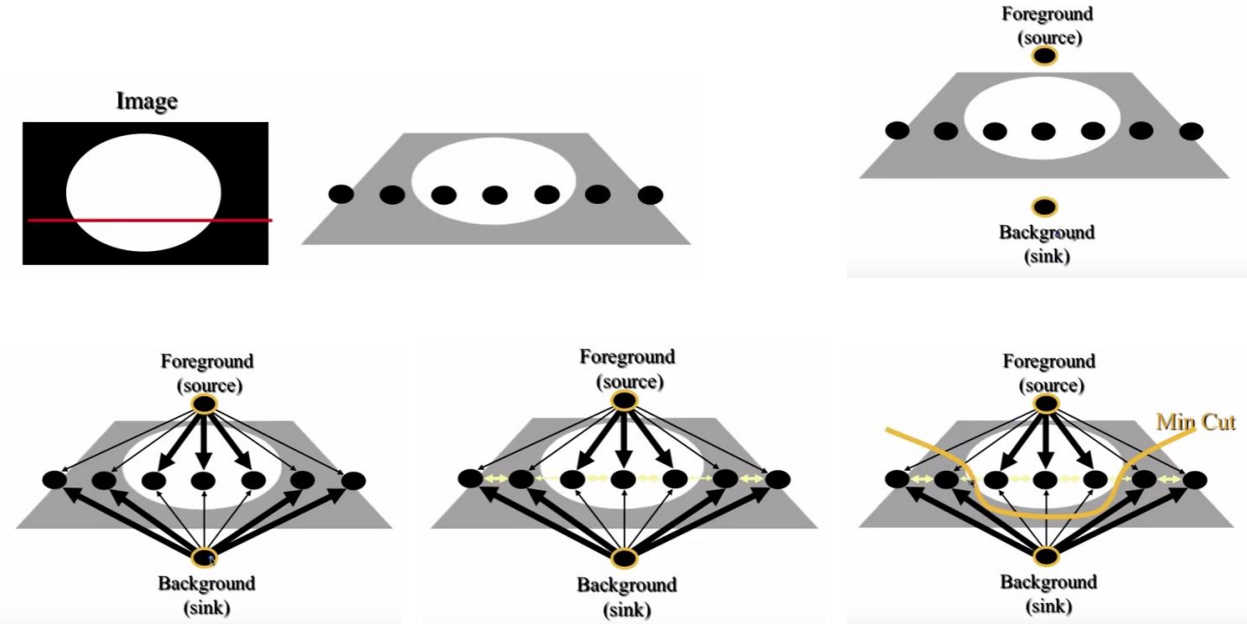

Fig. 6. **GraphCut Algorithm**. Each pixel of the image is treated as a node of a graph and two additional nodes are created, a *sink* and a *source*, representing the background and foreground, respectively. Min Cut Algorithms segments the image into foreground and background by partitioning the image along weak edges. Image Credits: Prof. Guillermo Sapiro's Coursera lecture

Lazy Snapping essentially uses GraphCut, but not by treating each individual pixel as a node as we just saw. It treats each segment obtained by Watershed Algorithm as a node. These '*presegments*' are often called superpixels (Figure 7) and have been illustrated below. As we mentioned earlier, this leads to a reduced complexity and hence the results are available at interactive speeds.

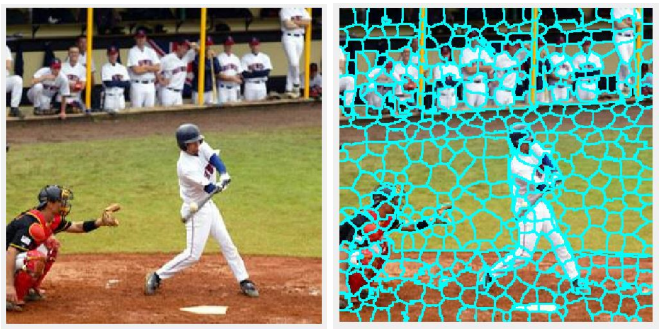

Fig. 7. **Superpixels of an Image**. Left: the original image; Right: The superpixels of the image which are obtained using Watershed Algorithm. Each of these superpixels are treated as a node in Lazy Snapping.

For the second part, the segmented foreground of the image is overlaid on the motion blurred version of the original image. The following figures are the output of the MATLAB® implementation which is made publicly available [here]. For reproduction of similar results, it must be noted that initial seeds that are provided by the user in the foreground and the background is a significantly important factor in deciding if the algorithm will give satisfactory results or not.

2.3. Examples from the Implementation

Fig. 8. **Examples from the Implementation**: The original images and their selective blurred versions. The foreground has been segmented out using Lazy Snapping and has been overlaid on the motion blurred original images.

3. GUI in Action

The following images show the GUI in action, for the two techniques, (1) Inpainting and (2) Selective Blur.

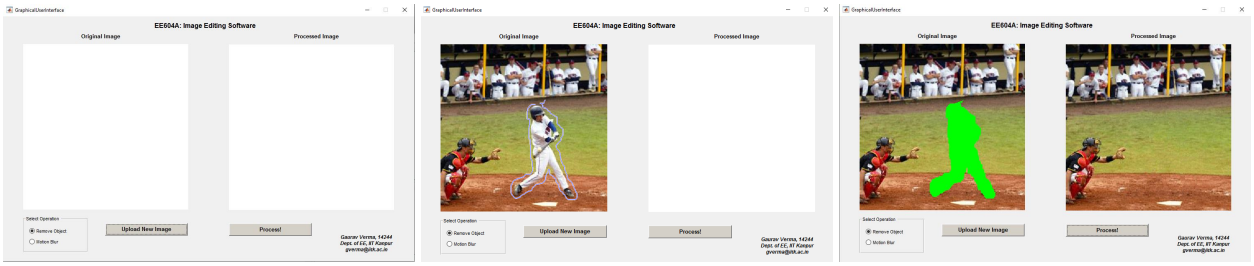

Fig. 9. **GUI in action for image inpainting**: The radiobutton corresponding to Image Inpainting is selected and the input image is uploaded. The object that is to be removed is selected using a freehand tool. Clicking on the *'Process'* button gives us the result of the implementation.

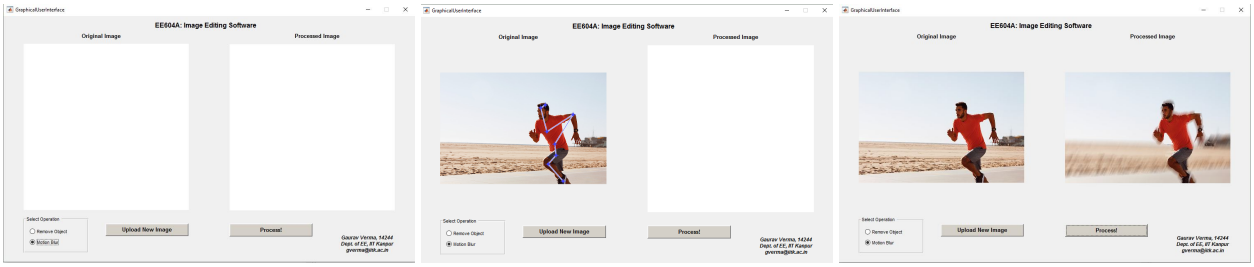

Fig. 10. **GUI in action for selective blurring** Select the seeds from the foreground and the background, followed by a click on *'Process!'* which will present the output image.

4. Acknowledgements

I would like to thank the Course Instructor for giving us the space to think that we can keep our style. I would also like to thank Ikuo Degawa, a Web Engineer in Tokyo as I referred to his implementation of inpainting whenever I found myself stuck.

References

1. Criminisi, Antonio, Patrick Perez, and Kentaro Toyama. "Object removal by exemplar-based inpainting." Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on. Vol.
2. Li, Y., Sun, J., Tang, C.K. and Shum, H.Y., 2004, August. Lazy snapping. In ACM Transactions on Graphics (ToG) (Vol. 23, No. 3, pp. 303-308). ACM.