Implementation of Region Filling and Object Removal by Exemplar-Based Image Inpainting

Luis Guzman

***Abstract*—This project consists of an implementation of the algorithm presented in Region Filling and Object Removal by Exemplar-Based Image Inpainting by Criminisi et al. [1]. The algorithm varies from other diffusion-based and texture-synthesis-based inpainting techniques in that it performs a direct copy of texture from the source region to the target region. This method avoids the blurry in-filling that is associated with diffusion-based techniques, and it can work for both large and small fill regions. The target region is filled in an order determined by its priority, which is a measure of the algorithm's confidence and the presence of texture patterns in the neighboring region. This allows prominent image structures such as lines and patterns to be propagated into the target region with high fidelity. My test images show that I successfully reproduced the results of the original paper through my implementation.**

I. Introduction

Image inpainting attempts to synthesize a patch of an image given an input image and a mask, which defines the patch to be filled. This technique is most often applied to removing unwanted objects from an image and can be a powerful tool for image restoration. In the typical use case, a user will take a photograph, designate objects to be removed, and then the algorithm is expected to fill in the area that the object occupies in a believable fashion. In many cases, the algorithm presented by Criminsi et al. can create a filled-in region that is indistinguishable from the rest of the image.

An important factor to the performance of these algorithms is contextual awareness. Humans tend to focus on patterns within an image and can easily detect when those patterns are broken [2]. For this reason, preserving the patterns that exist at the infilling boundary is essential to creating a convincing filled region. This algorithm prioritizes regions that have strong patterns near the boundary so that the consistency of these patterns is preserved.

Early attempts at image inpainting use diffusion to propagate patterns inward [3], [4], [5]. These methods propagate regions of similar color (known as isophotes) inwards using the Navier-stokes equation. This method, adopted from the study of heat transfer, uses the pixel colors on the boundary as the "boundary condition" and allows those colors to naturally propagate inwards. Diffusion methods can preserve patterns such as lines and are quite effective at filling small regions, but replacing large objects can lead to a blurry result.

Texture-synthesis methods aim to improve on this by generating a unique texture to fill the target region [6], [7], [8]. The goal of these methods is to use the known pixel values, and calculate the best-fit texture for the unknown region. Although some methods generate the texture from scratch, the most believable results come from filling with texture patches (exemplars) from the source image. These methods can produce a believable texture, but the structure of the isophotes is usually lost. The algorithm proposed by Criminsi et al. combines the strengths of these two methods, and can create believable exemplar-based textures while also preserving the isophotes at the target boundary.

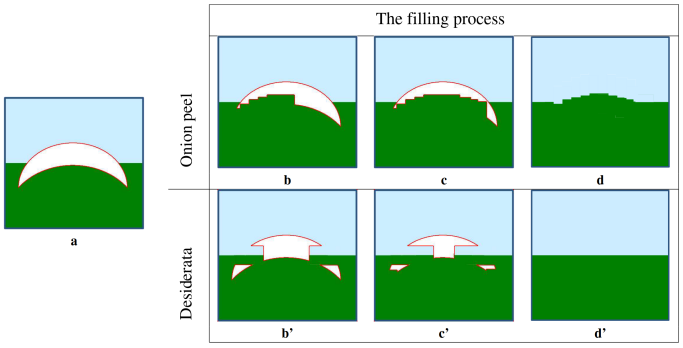

Figure 1. Fill order is an essential part of this algorithm. Using a typical "onion peel" order results in linear structures being lost inside the target region. Filling the areas with the strongest lines first results in correct structure propagating the fill region.

The current state-of-the-art in image inpainting uses Generative Adversarial Networks (GANs) to fill in the missing regions [10], [11], [12], [13]. These methods consist of two deep neural networks: one generator and one discriminator. The generator takes in an image with missing portions (as specified by the object removal mask) and attempts to generate a new image where the missing portions are filled in. During training, the discriminator then takes the filled-in image and outputs a measure of how believable the image is. Through joint training, these two networks can learn to generate images from the missing portions that are indistinguishable from the ground truth to the human eye. In this project, I won't be examining these methods, but it's worth noting that they can offer significantly better performance than the non-machine learning method implemented here.

II. Methods and Theory

The algorithm can be summarized as follows: First, identify the target boundary, which is the line separating the target region and the source image. Next, compute the priority of each pixel on the boundary and select the boundary point with the highest priority. Lastly, copy the exemplar from the source that best matches the known pixels around point and update the priority values. These steps are then repeated until the entire target region has been filled. All calculations are explained in

detail in the following sections.

Before beginning the iteration, Criminsi et al. specifies that the size of the target region Ψ_p must be chosen. This area is 9x9 by default but should be chosen to be larger than the smallest resolvable texture element (texel) in the image. To standardize my implementation, I used the rule

$$\texttt{texel\_size} = \min(\texttt{img.shape})/30$$

and I round this value to the nearest odd number. This method results in a 9x9 texel when a 256x256 image is used, and a correspondingly larger texel for larger images. This is an improvement that I made over the original implementation to minimize the need for user input.

In order to determine the target boundary (denoted $\delta\Omega$), I chose to use `cv2.findContours` for simplicity. Since this step is not the focus of the algorithm and well-known edge detection tools exist, I chose not to implement this from scratch. Specifically, the contour filtering of OpenCV is essential because I had to ensure the contour is continuous and has width of one pixel for the following step. The result of this operation is a dense list of pixel values of shape `[numContours, numPoints, 1, 2]`, where the last axis contains the (u, v) pixel locations of the boundary.

As shown in figure 1, an essential aspect of this algorithm is the filling order. Criminsi et al. proposes a measure called the priority to determine this. Given a point p on the target boundary $\delta\Omega$, the priority P is the product of the confidence C and the data D terms:

$$P(\mathbf{p}) = C(\mathbf{p}) * D(\mathbf{p})$$

where

$$C(\mathbf{p}) = \frac{\sum_{\mathbf{q}\in\Psi_\mathbf{p}\cap(\mathcal{I}-\Omega)} C(\mathbf{q})}{|\Psi_\mathbf{p}|}, \qquad D(\mathbf{p}) = \frac{|\nabla I_\mathbf{p}^\perp \cdot \mathbf{n_p}|}{\alpha}$$

$|\Psi_p|$ is the area of the target region Ψ_p, α is a normalization factor of 255, n_p is the vector normal to the fill front $\delta\Omega$, and ∇I_p is the unit vector in the direction of the isophote from the region $I \cap \Psi_p$. Figure 2 specifies all the notation pictorially.

The confidence term essentially measures how confident the algorithm is in its predictions. The confidence term C is initialized to be 0 for points in the target region and 1 for points in the source region. Because the target region is filled inwards from the outer boundary, the confidence values naturally decay as the algorithm precedes further inwards. Also, the confidence value is higher when there are more source pixels in the target window, so this term prioritizes convex regions that have more known pixels than unknown pixels. This way, the algorithm can start by filling the easier region, and fill the hardest regions last to avoid propagating potential mistakes.

The data term is a measure of the linear structure in the target window. This term prioritizes pixels that have isophotes protruding into the target window. Filling these areas first allows the algorithm to maintain lines and other structures at the boundary of the target region. Figure 3 shows how the data term prioritizes these structures and leads to continuous linear structures.

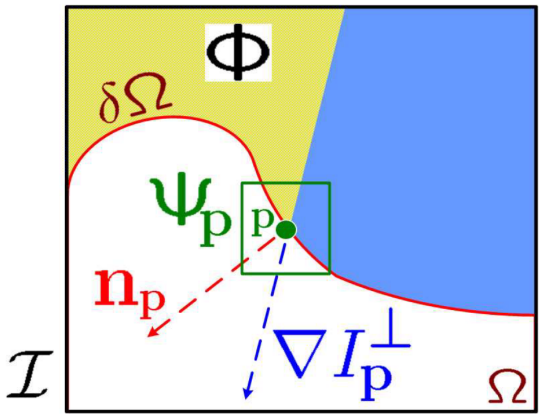

Figure 2. Notation diagram. Given the patch Ψ_p, n_p is the normal to the contour $\delta\Omega$ of the target region Ω and ∇I_p is the isophote (direction and intensity)at point p. The entire image is denoted with I. Figure taken directly from Criminisi et al.

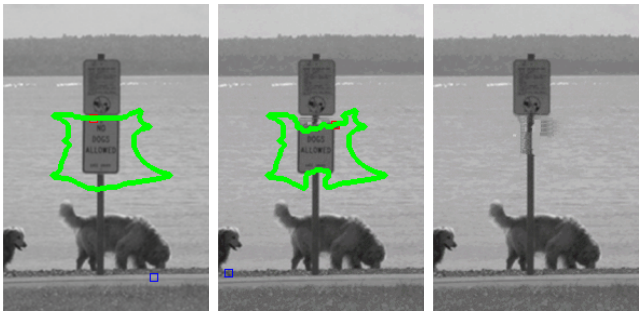

Figure 3. Preservation of linear structures. The original image is on the left. The data term prioritizes the regions that include the pole, since those regions have the strongest isophotes. These regions are filled first, so that the pole can be propagated continuously through the target region. The middle image is 40 iterations in, and the right image is the final result. Any ghosting artifacts in the final result are due to the low image resolution, since the provided resolution (191x284) means that the gap between the two signs is only one pixel wide.

In order to calculate the normal n_p of the boundary, I use

$$n_{p_i} = \frac{(p_i - p_{i-1}) \times (0, 0, 1)}{|(p_i - p_{i-1}) \times (0, 0, 1)|}$$

where p_i is the 3D pixel coordinate $(u, v, 0)$. Criminisi et al. suggest using a gaussian filter to ensure that aliasing of the boundary don't cause the normals to change too quickly. I used a moving average filter of size 2 to accomplish this. A visualization of the calculated normals is given at the end of this paper, in figure 9.

The isophote ∇I_p is calculated similarly. First, I get the gradient image $\nabla\Psi_p$ of the target area (9x9 by default) surrounding point p. The most prominent linear structure is then $\arg\max(\nabla\Psi_p)$. Since this vector is a gradient, it will be aligned perpendicular to the strongest line in the image. I then rotate it to align with the linear structure with

$$\nabla I_p = \frac{\arg\max(\nabla\Psi_p) \times (0, 0, 1)}{|\arg\max(\nabla\Psi_p) \times (0, 0, 1)|}$$

where again the all vectors are in 3D coordinates $(u, v, 0)$. The amplitude of this vector indicates how strong the linear

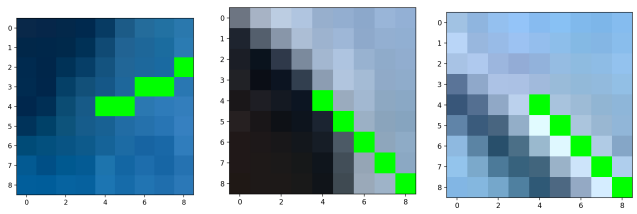

Figure 4. Result of the isophote calculation. The isophote unit vector (green) is aligned with the most prominent lines in the image

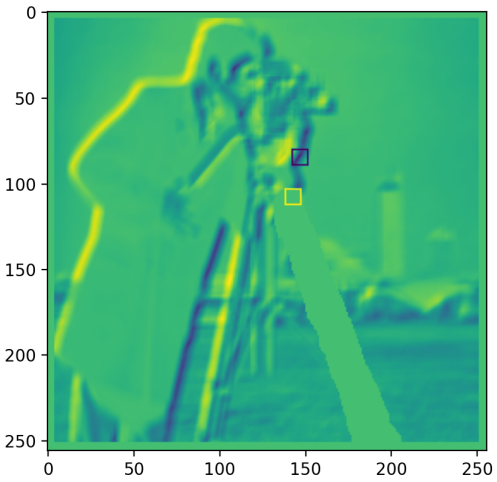

Figure 5. The distance image. Higher distances are displayed in yellow and smaller are in blue. The target window is shown in the yellow box and the best-fit exemplar is the blue box. Note how the best-fit exemplar is centered around the point with the smallest disance. These values will be copied into the yellow box and will propagate the linear structure into the target region.

structure are in the target window. I found that I achieved better performance by using the 90th percentile of the gradient, rather than the maximum, to filter out any effects from noise. Figure 4 shows the results of my isophote calculation. Note how the green isophote vector aligns with the most prominent linear structure in the window.

Once the point p with highest priority has been chosen, the algorithm searches for the best-match exemplar to fill the target region with. The best-fit is defined as

$$\Psi_{\hat{q}} = \arg\min_{\Psi_q \in \Phi} d(\Psi_{\hat{p}}, \Psi_q)$$

where d is the sum of squared distances (SSD) of the two image windows in the CIE Lab color space. Figure 5 provides a visualization of this distance metric. The CIE Lab color space is used because distances are more meaningful than with RGB values due to the perceptual uniformity property. This is the most expensive calculation of the entire algorithm since every pixel window in the source image is considered and compared against the target window. Once the best-fit is found, the pixel values are directly copied from the source window to any unknown pixels in the target region.

After the pixel values are filled, the confidence value is updated for every pixel in the target window. The update rule

- Extract the manually selected initial front $\delta\Omega^0$.
- Repeat until done:

1a. Identify the fill front $\delta\Omega^t$. If $\Omega^t = \emptyset$, exit.
1b. Compute priorities $P(\mathbf{p}) \quad \forall \mathbf{p} \in \delta\Omega^t$.
2a. Find the patch $\Psi_{\hat{\mathbf{p}}}$ with the maximum priority, *i.e.*, $\hat{\mathbf{p}} = \arg\max_{\mathbf{p} \in \delta\Omega^t} P(\mathbf{p})$.
2b. Find the exemplar $\Psi_{\hat{\mathbf{q}}} \in \Phi$ that minimizes $d(\Psi_{\hat{\mathbf{p}}}, \Psi_{\hat{\mathbf{q}}})$.
2c. Copy image data from $\Psi_{\hat{\mathbf{q}}}$ to $\Psi_{\hat{\mathbf{p}}}$ $\forall \mathbf{p} \in \Psi_{\hat{\mathbf{p}}} \cap \Omega$.
3. Update $C(\mathbf{p}) \quad \forall \mathbf{p} \in \Psi_{\hat{\mathbf{p}}} \cap \Omega$

Figure 6. Pseudocode of the inpainting algorithm

is

$$C(\mathbf{p}) = C(\hat{\mathbf{p}}) \qquad \forall \mathbf{p} \in \Psi_{\hat{p}} \cap \Omega$$

and this specifies that every pixel in the target window gets assigned the confidence previously held by the boundary point p. Each step is then repeated until the entire target region has been filled. The algorithm pseudocode is shown in figure 6.

III. Results

In order to test the correctness of my implementation, I tested my algorithm on the same images from the original paper. As shown in figure 7, my implementation produces very similar results as the original paper. Specifically, the isophotes of the pole are correctly propagated through the target region, which indicates that my calculation of the priority and filling order are correct. In each image, the target region is replaced with a believable texture that is contextually consistent. Note that the images may vary slightly from the original paper due to slight differences in the image mask and the target window size, which was not specified for all images.

I have ordered the images in progression of best-to-worst. In the images of the river and the man jumping, my implementation produced better reconstructions than those in the original paper. The riverbank in the first image appears to be more continuous than the jagged boundary shown in Criminisi et al. Additionally, the shoreline of the lake in the second image does not protrude into the lake in my implementation, and the blurry area above the barn is absent.

To my eye, the "Japanese animation" image is of equal quality to the original paper. In both my image and Criminisi et al's, there are areas that appear blurry or seem to have artifacts due to the complex background.

In the last two images, my implementation achieved worse results than the original. In the image of the dog, there are some ghosting artifacts near the sign post. I believe this is due to the low resolution of the image. All images were taken directly from Criminisi et al.'s paper, so I am working with a lower resolution image. This means that the algorithm has less data to compare with during the exemplar search, so the exemplar-matching does not find as good of a match.

The image of the woman in front of water was my worst performing image. The algorithm correctly replaces the texture of the water behind the woman, but it seems to have trouble here connecting the lines. I believe this could be due to errors in my normal calculation, because if the normals of the image are off, the lines could get slightly skewed. However, the lines are propagating into the target region correctly, so it could

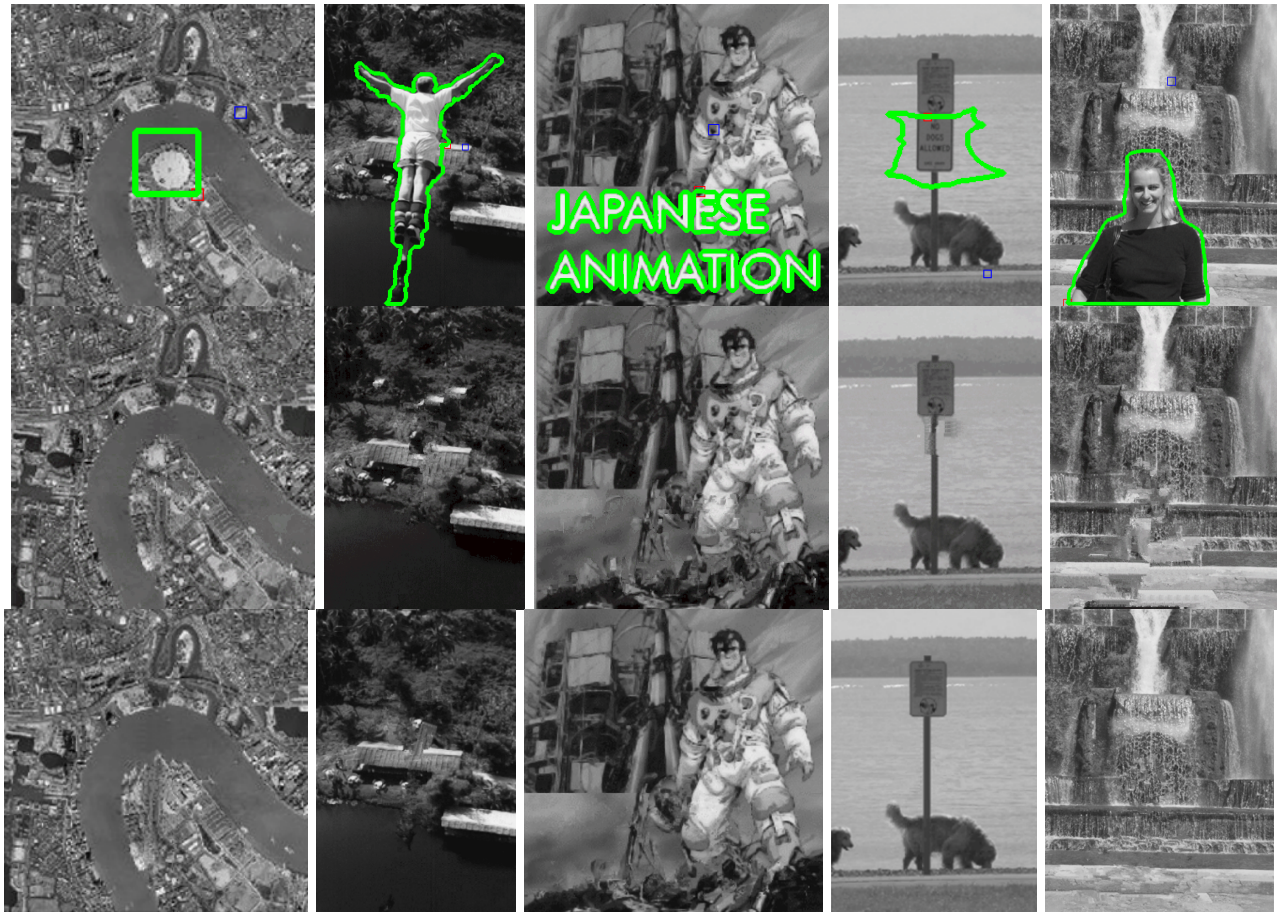

Figure 7. Results of the image inpainting algorithm. Input and mask are shown in the top row. My implementation is the middle row. Examples from Criminisi et al. are shown on the bottom.

just be that the target region is too wide for them to connect reliably.

I considered using some quantitative measure such as mean-squared-error to determine the quality of my inpainting results, but due to the lack of ground-truth images and the fact that Criminisi et al. also did not include quantitative results, I have left this aspect to future work. In addition to my tests on the original paper's images, I used images from the Places 365 dataset [9] for further testing (shown in figure 8). These images enabled me to test on large color images where the background may be more complex. The image size of this dataset is approximately 1100x800, so it was a challenge to remove objects that appear much larger than the images in figure 7.

In the image of the hotel lobby, the reconstruction is very good close to the target boundary. The blue lines are correctly propagated into the target region and the texture of the tiles appears to be correct. However, the reconstruction breaks down near the center of the images and especially around the chair because there are not any exemplars that match this area well. This is an example of an image where GANs would produce much better results because they would be able to recognize the portion of the chair and fill in the rest, even though a similar chair does not exist in the original image.

In hay bail image, the algorithm correctly connects the green lines of the trees and the bright yellow of the wheat. There are some artifacts near the image center, again due to the large and complex image, but overall I think it is a decent reconstruction.

IV. Discussion

Although my implementation achieved similar overall performance to the original paper, I'd like to use this section to discuss some of the difference and challenges I faced while implementing this algorithm. The first challenge was to minimize artifacts in the reconstructed image. An example of these artifacts can be seen in the image of the woman and fountain in figure 7. An additional type of artifact is the ghosting that can be seen in the dog image in the same figure. The most common cause for these artifacts is incorrect fill order. While implementing the algorithm, I found many bugs in my calculations for the confidence and data terms. Oftentimes, the calculation would appear to work well for one image, just to be broken by the next test image. I visualized this calculation (shown in figure 10) as much as possible to ensure that it's working properly, but there's always a chance that I missed something during my testing.

An additional cause of artifacts is an inadequate texel window size. I attempted to automate the process of selecting this size, but I found that my automated solution does not work for some images. Depending on the amount of detail in the

Figure 8. Results on additional images. Input and mask are shown in the top row. Results of my implementation is the bottom row.

images, the end user may still need to adjust the window size to eliminate artifacts. For example, in the image of the hotel, I had to make the target window smaller so that the people weren't unnaturally replicated inside the target window.

Another area that I spent time improving is the case of objects that touch the border of the image. In this scenario, the exemplar comparison fails because there is no data outside the image boundary to compare to. In this case, I set the confidence value to zero because there are no points to sum over. This results in the edge points being filled last, and therefore the edge does not significantly impact the reconstruction of the entire target. A downside of this approach is that there can be a small sliver of the original object left over (as seen in the woman and fountain image), but I accepted this compromise, as dealing with the edge case in other means would require significant modifications to my implementation.

An area of evaluation for any implementation is the algorithm's time-performance. As mentioned in the methods section, finding the best-fit exemplar is the most time-consuming step, so I spend some time attempting to optimize this process. For the river image in figure 7, Criminisi et al. Report that their algorithm fills the region in 2 seconds. In comparison, my implementation took 90 seconds to fill the same region. Additionally, to fill image of the man jumping, the original paper claims an 18 second runtime, and my algorithm took 8 minutes.

Before discussing how I aimed to improve the runtime efficiency, I'd like to note that my implementation was not multithreaded. I did not explore this option since I have not had experience with parallelizing code in the past, but this could be an additional reason why my implementation takes longer to run than Criminisi's. If the exemplar search were parallelized, the code would take approximately a minute to run on an 8 thread processor, which is much closer to Criminisi's reported time.

I attempted to improve on this by vectorizing the exemplar searching calculation. My first attempt at this was to create a 4D array that stored the neighborhood around each pixel. Each image would then be of shape `[imgH, imgW, 3 (channels), size`^{2}`]` and the best-fit exemplar would be

$$\arg\min_{i,j}\left(\sum_{k,l}(I[i,j,k,l]-\Psi_p[k,l])^2\right)$$

where $I[i,j,k,l] = \Psi_q[k,l]$ at point $q = [i,j]$. This worked well for small images, but I ran into an out-of-memory error when running on images larger than 1000x1000. After attempting to decrease the memory usage by saving fewer values, the final method was not significantly faster than a brute-force search, due to having to update my 4D matrices in each iteration. I opted to instead define a stride as

$$\texttt{stride} = \max(\lfloor \texttt{size}/9 \rfloor, 1)$$

The max function ensures that the stride is greater than or equal to one, and the floor and division operators scale the stride according to the texel size. For texels up to size 18, a stride of 1 is used, so this value only speeds up the runtime on larger images. Other algorithms for image inpainting can take 1-2 hours to run on a 384x256 image, so my implementation still makes a considerable improvement on those.

V. Conclusion

In this project report I successfully implemented the image inpainting algorithm presented by Criminisi et al. My implementation is able to fill the target region with believable textures that are contextually consistent so that oftentimes they are indistinguishable to the human eye. This texture is

generated by filling the target region in order defined by each pixels calculated priority, selecting the best-fit exemplar from the source region, and performing a direct copy of exeplars from the source to the target region. My implementation has a reasonable runtime, running much faster than previous image inpainting algorithms, although being slower than the implementation in the original paper. Further work includes improving this runtime by parallelizing the exemplar search, and potentially using machine learning techniques to improve the inpainting quality and to eliminate the need for the user to select objects for removal. Overall, my implementation reproduced the visual fidelity of the results in the original paper, and proved effective on additional test images from the Places 365 dataset.

VI. Links

Project Video: https://youtu.be/wLUDYjY6nEg
Github: https://github.com/luigman/image-inpainting

References

[1] A. Criminisi, P. Perez and K. Toyama, "Region filling and object removal by exemplar-based image inpainting," in IEEE Transactions on Image Processing, vol. 13, no. 9, pp. 1200-1212, Sept. 2004, doi: 10.1109/TIP.2004.833105.

[2] M. J. Tarr, "Visual pattern recognition," in Encyclopedia of psychology, Washington, D.C.: Oxford [Oxfordshire]; New York: American Psychological Association; Oxford University Press, 2000, pp. 66–71.

[3] C. Ballester, V. Caselles, J. Verdera, M. Bertalmio, and G. Sapiro, "A variational model for filling-in gray level and color images," in Proc. Int. Conf. Computer Vision, Vancouver, BC, Canada, June 2001, pp. 10–16.

[4] M. Bertalmio, A. L. Bertozzi, and G. Sapiro, "Navier-stokes, fluid dynamics, and image and video inpainting," in Proc. Conf. Comp. Vision Pattern Rec., Dec. 2001, pp. 355–362.

[5] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester, "Image inpainting," in Proc. ACM Conf. Comp. Graphics (SIGGRAPH), New Orleans, LA, July 2000, http://mountains.ece.umn.edu/ guille/inpainting.htm, pp. 417–424

[6] A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer," in Proc. ACM Conf. Computer Graphics (SIGGRAPH), Aug. 2001, pp. 341–346.

[7] A. Efros and T. Leung, "Texture synthesis by nonparametric sampling," in Proc. Int. Conf. Computer Vision, Kerkyra, Greece, Sept. 1999, pp. 1033–1038.

[8] W. T. Freeman, E. C. Pasztor, and O. T. Carmichael, "Learning low-level vision," Int. J. Comput. Vis., vol. 40, no. 1, pp. 25–47, 2000.

[9] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva and A. Torralba, "Places: A 10 Million Image Database for Scene Recognition," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 40, no. 6, pp. 1452-1464, 1 June 2018, doi: 10.1109/TPAMI.2017.2723009.

[10] K. Nazeri, E. Ng, T. Joseph, F. Qureshi, and M. Ebrahimi, "Edge-Connect: Structure Guided Image Inpainting using Edge Prediction," 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), 2019.

[11] Yu, Jiahui, et al. "Generative Image Inpainting with Contextual Attention." ArXiv:1801.07892 [Cs], Mar. 2018. arXiv.org, http://arxiv.org/abs/1801.07892.

[12] Yu, Jiahui, et al. "Free-Form Image Inpainting with Gated Convolution." ArXiv:1806.03589 [Cs], Oct. 2019. arXiv.org, http://arxiv.org/abs/1806.03589.

[13] Ulyanov, Dmitry, et al. "Deep Image Prior." International Journal of Computer Vision, vol. 128, no. 7, July 2020, pp. 1867–88. arXiv.org, doi:10.1007/s11263-020-01303-4.

Figure 9. Visualization of the normal calculation step.

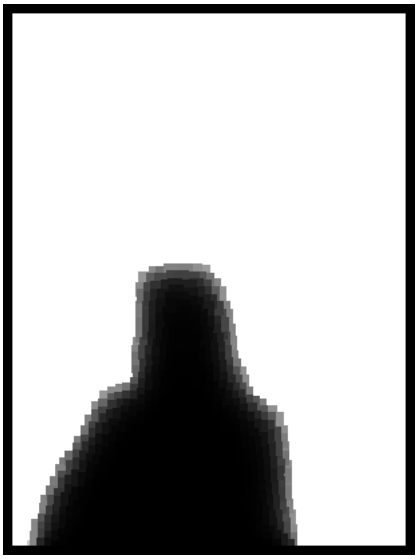

Figure 10. Visualization of the confidence calculation. Notice how the values decay as you move further into the target region.