Criminisi Inpainting

Release 0.00

David Doria

February 4, 2011

Rensselaer Polytechnic Institute, Troy NY

Abstract

This document presents a system to fill a hole in an image by copying patches from elsewhere in the image. These patches should be a good continuation of the hole boundary into the hole. The patch copying is done in an order which attempts to preserve linear structures in the image. This implementation is based on the algorithm described in "Object Removal by Exemplar-Based Inpainting" (Criminisi et. al.).

The code is available here: https://github.com/daviddoria/Inpainting

Contents

1 Introduction

This document presents a system to fill a hole in an image by copying patches from elsewhere in the image. These patches should be a good continuation of the hole boundary into the hole. The patch copying is done in an order which attempts to preserve linear structures in the image. This implementation is entirely based on the algorithm described in [1]. There are many subtle details of the implementation which are explained throughout this paper.

2 Terminology

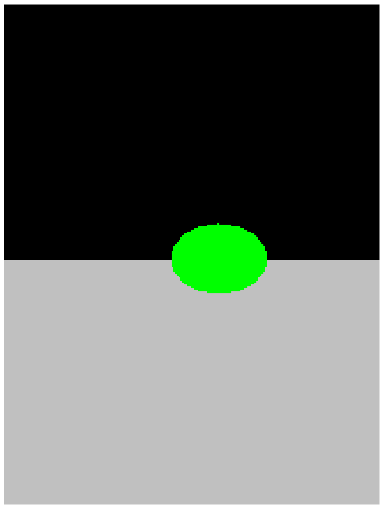
Throughout this document, the "source region" is the portion of the image which is known (is not part of the hole) at the beginning. The "target region" is the current hole to be filled.

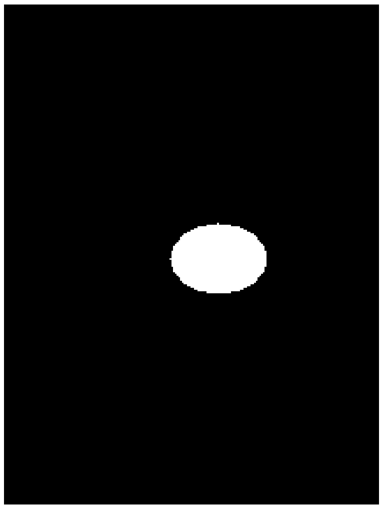
3 Algorithm Overview

The inputs to the algorithm consist of an image and a binary mask the same size as the image. Non-zero pixels in the mask indicate the region of the image which is to be considered the hole to inpaint/complete. Throughout this paper, we have colored the region in the input image corresponding to the hole bright green. This color irrelevant - we have done this only to make it obvious to tell if any part of the hole remains after inpainting (it should not). In practice, the input image need not be modified.

4 Algorithm Synthetic Demonstration

Figure 1 shows a synthetic demonstration of the algorithm. The image consists of a black region (top) and a gray region (bottom). This simple example is used for testing because we know the result to expect - the dividing line between the black region and gray region should be continued smoothly.


(a) Image to be filled. The region to be filled is shown in bright green.


(b) The mask of the region to inpaint.

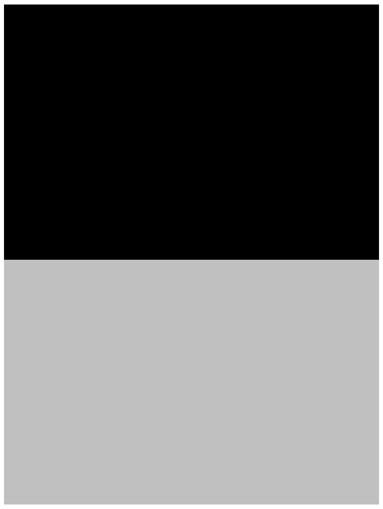
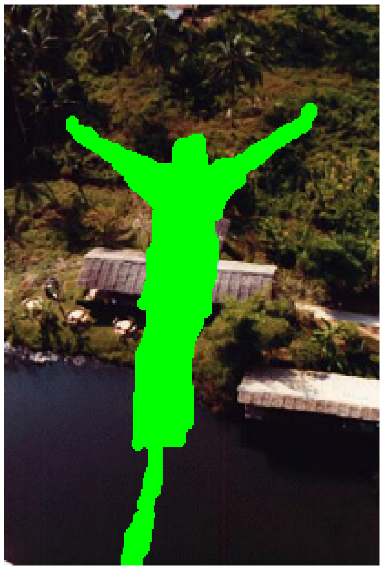
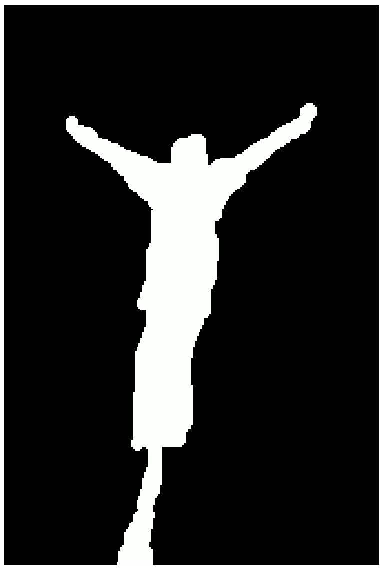

(c) The result of the inpainting.

Figure 1: Synthetic Demonstration

5 Algorithm Realistic Demonstration

Figure 2 shows a real example of the algorithm. This result shows the typical quality of inpainting that the algorithm produces.


(a) Image to be filled. The region to be filled is shown in bright green.


(b) The mask of the region to inpaint.

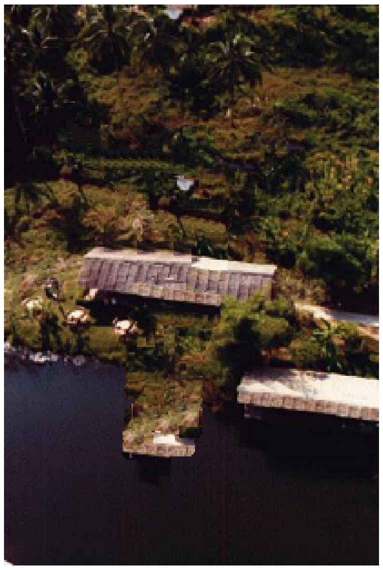

(c) The result of the inpainting. This took about 30 seconds on a P4 3GHz processor with a 206x308 image and a patch radius = 5.

Figure 2: Realistic Demonstration

6 Algorithm Structure

An overview of the algorithm is:

- Initialize:
  - Read an image and a binary mask. Non-zero pixels in the mask describe the hole to fill.
  - Set the size of the patches which will be copied. (Typically an 11x11 patch (patch radius = 5) is used).
  - Locate all patches of the image which are completely inside the image and completely in the source region. These are stored as an $std :: vector < itk :: ImageRegion < 2 >>$ named SourcePatches.
- Main loop:
  - Compute the priority of every pixel on the hole boundary (see Section 7.1)
  - Determine the boundary pixel with the highest priority. We will call this the target pixel. The region centered at the target pixel and the size of the patches is called the target patch.
  - Find the SourcePatch which best matches the portion of the target patch in the source region.
  - Copy the corresponding portion of the source patch into the target region of the target patch.
  - Repeat until the target region consists of zero pixels.

7 Algorithm Details

The two main parts of the algorithm are

1. Determine the priority of each boundary pixel. This determines the order with which the hole is filled.
2. Find the best matching patch to the patch around the pixel with the highest priority.

7.1 Priorities

The priority $P(p)$ of a pixel p is given by the product of a Confidence term $C(p)$ and a Data term $D(p)$.

$$P(p) = C(p)D(p) \tag{1}$$

The original author describes the terms:

“C(p) may be thought of as a measure of the amount of reliable information surrounding the pixel ’p’. The intention is to fill first those patches which have more of their pixels already filled, with additional preference given to pixels that were filled early on (or that were never part of the target region).”

“D(p) is a function of the strength of the isophotes hitting the front at each iteration. This term boosts the priority of a patch that an isophot ”flows“ into. This factor is of fundamental importance in our algorithm because it encourages linear structures to be synthesized first, and, therefore propagated securely into the target region.”

Confidence Term

The confidence term is computed as:

$$C(p) = \frac{\sum \text{Confidences of patch pixels in the source region}}{\text{Area of the patch}} \tag{2}$$

To initialize, the confidence inside the hole is set to 0 and the confidence outside the hole is set to 1.

Data Term

The data term is computed as:

$$D(p) = \frac{\text{dot}(\text{isophote}, \text{boundary normal})}{\alpha} \tag{3}$$

α is a normalization factor that should be set to 255 for grayscale images, but that value also seems to work well for RGB images.

No initialization is necessary because this term is not recursive - it can be computed from the data directly at each iteration.

7.2 Patch Matching

To compare two patches (a source patch and a target patch), we compute the normalized sum of squared differences between every pixel which is in the source region of target patches (all pixels of the SourcePatch are in the source region by definition).

8 Implementation Details

8.1 Isophotes

An isophotes is simply a gradient vector rotated by 90 degrees. It indicates the direction of “same-ness” rather than the direction of maximum difference. There is a small trick, however, to computing the isophotes. We originally tried to compute the isophotes using the following procedure:

- Convert the RGB image to a grayscale image.
- Blur the grayscale image.
- Compute the gradient using itkGradientImageFilter.
- Rotate the resulting vectors by 90 degrees.
- Keep only the values in the source region.

This procedure produces the gradient magnitude map shown in Figure 3.

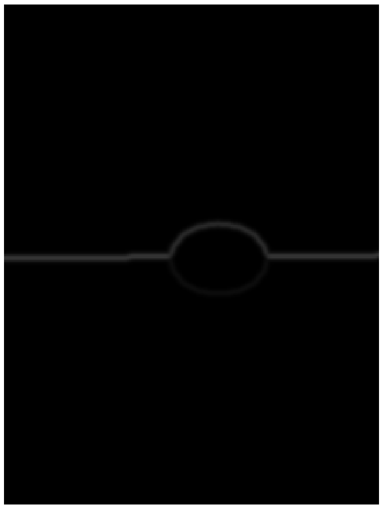
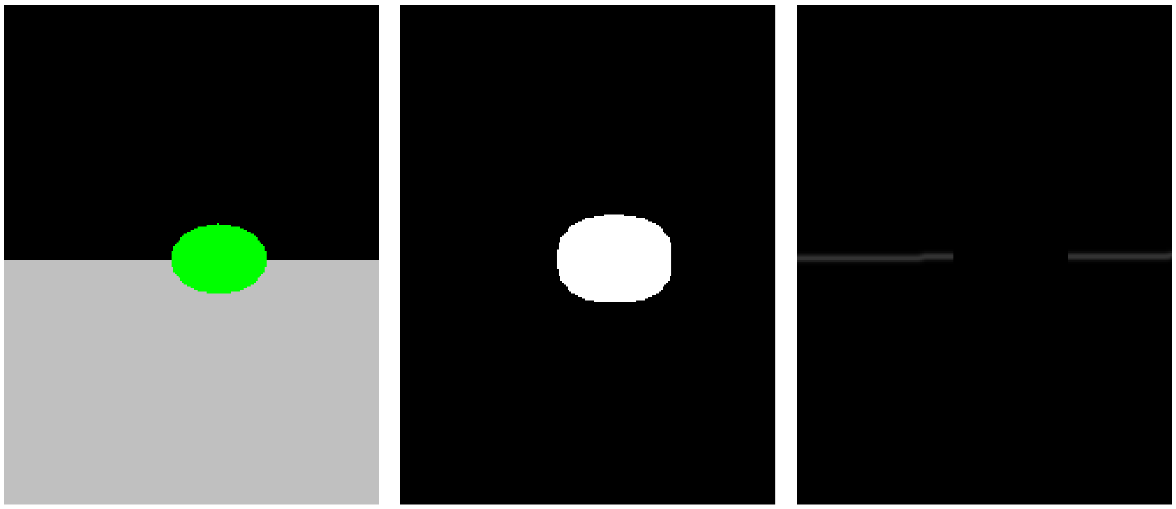


Figure 3: Result of naively computing the image gradient.

The high values of the gradient magnitude surrounding the target region are very troublesome. The resulting gradient magnitude image using this technique is sensitive to the choice of the pixel values in the target region, which we actually want to be a completely arbitrary choice (it should not affect anything). More importantly, the gradient plays a large part in the computation of the pixel priorities, and this computation is greatly disturbed by these erroneous values. Simply ignoring these boundary isophotes is not an option because the isophotes on the boundary are exactly the ones that are used in the computation of the Data term. To fix this problem, we immediately dilate the mask specified by the user. This allows us to compute the isophotes as described above, but now we have image information on both sides of the hole boundary, leading to a valid gradient everywhere we need it to be. Figure 4 shows the procedure for fixing this problem.



(a) Image to be filled. The target region is shown in green.

(b) The dilated mask.

(c) The gradient magnitude with pixels in the new (enlarged) target region set to zero.

Figure 4: Procedure for fixing the erroneous gradient problem.

As you can see, this gradient magnitude image is exactly what we would expect.

8.2 Boundary Normals

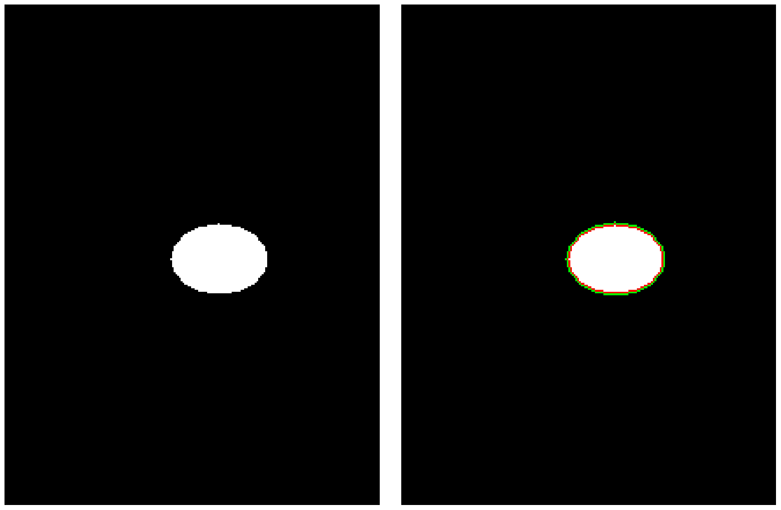
There are two things to be careful with when computing the boundary normals: computing the normals only on the one pixel thick boundary, and using the correct side of the masked region as the boundary.

Computing boundary normals only on the one pixel thick boundary

If we compute the normals directly on the binary mask, the set of resulting vectors are too discretized to be of use. Therefore, we first blur the mask. However, the gradient of the blurred mask results in non-zero vectors in the gradient image in many more pixels (a "thick" boundary) than the gradient of the original mask (a single pixel "thin" boundary). Therefore, we must mask the gradient of the blurred mask to keep only the pixels which would have been non-zero in the original mask gradient.

Using the correct side of the masked region as the boundary

There are two potential boundaries that can be extracted from a masked region - the "inner" boundary and the "outer" boundary. As shown in Figure 5, the inner boundary (red) is composed of pixels originally on the white (masked) side of the blob, and the outer boundary (green) is composed of pixels originally on the black (unmasked) side of the blob. It is important that we use the outer boundary, because we need the boundary to be defined at the same pixels that we have image information, which is only in the source (black/unmasked) region.



(a) The inner boundary.

(b) Outer boundary (green) and inner boundary (red).

Figure 5: Inner vs Outer Boundary of a Region

Code Snippet

The CriminisiInpainting class must be instantiated using the type of image to be inpainted. Then the patch radius must be set, the image and mask provided, and the Inpaint() function called.

```
  CriminisiInpainting<ImageType> Inpainting;
  Inpainting.SetPatchRadius(5);
  Inpainting.SetImage(imageReader->GetOutput());
  Inpainting.SetInputMask(maskReader->GetOutput());
  Inpainting.Inpaint();

  ImageType::Pointer result = Inpainting.GetResult();
```

If you would like to see what happens at every step of the algorithm, you can use:

```
Inpainting.SetWriteIntermediateImages(true);
```

NOTE: Several images are output at each iteration - these files could be quite large!

References

[1] A. Criminisi, P. Perez, K. Toyama, *Object Removal by Exemplar-Based Inpainting*. Computer Vision and Pattern Recognition 2003 1