A Patch-Based Inpainting Framework

Release 0.00

David Doria

November 15, 2012

Rensselaer Polytechnic Institute, Troy NY

Abstract

This document describes 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 image outside the hole boundary into the hole. The implementation is very generic, allowing the develop to select or easily add new methods for the patch priority order, patch comparison function, and other parameters of the algorithm.

The "basic" algorithm is called ClassicalImpageInpainting and 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/PatchBasedInpainting

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3250]

Contents

1 Introduction

This document describes 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.

2 Dependencies

This code makes heavy use of multiple libraries:

- VTK >= 6.0
- ITK >= 4.2
- Boost >= 1.51
- CMake >= 2.8.6
- Qt >= 4.8

The code is also organized into git submodules. These are included if you clone with:

```
git clone --recursive https://github.com/daviddoria/PatchBasedInpainting.git
```

or

```
git clone https://github.com/daviddoria/PatchBasedInpainting.git
git submodule update --init --recursive
```

The required submodules are:

- Mask
- ITKHelpers (via Mask)
- Helpers (via Mask ITKHelpers)
- BoostHelpers
- CMakeHelpers
- ITKQtHelpers
- ITKVTKCamera
- ITKVTKHelpers
- QtHelpers
- VTKHelpers

3 Terminology

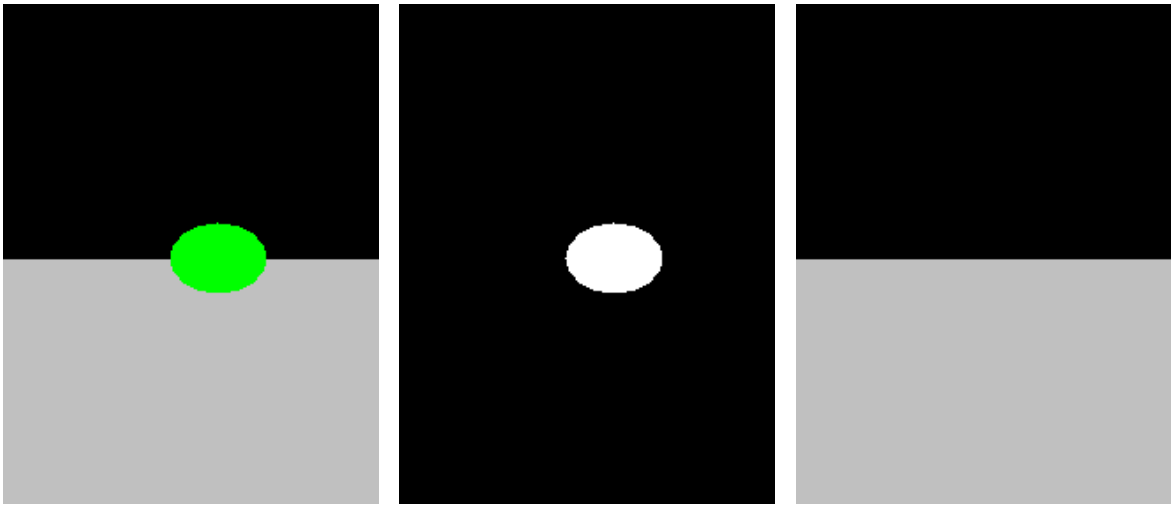
Throughout this document, the "source region" is the portion of the image which is known (is not part of the hole) at the beginning. The "target region" is the current hole to be filled.

4 Basic Algorithm Overview

The inputs to the algorithm consist of an image and a binary mask the same size as the image. We use a custom Mask class to describe the hole (https://github.com/daviddoria/Mask). 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), and for easier debugging to ensure these pixels are not used in any part of the computation. In practice, the input image need not be modified.

5 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

6 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

7 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 **??**)
  - 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 source Pptch 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.

8 Priority Functions

The priority function is used to determine which target patch to fill next. We provide several such functions.

8.1 PriorityCriminisi

The priority term described in [1] is is given by the product of a Confidence term $C(p)$ and a Data term $D(p)$. This priority function attempts to both continue linear structures sooner rather than later, and fill patches where a larger number of the pixels in the patch are already filled.

8.2 PriorityConfidence

This priority function is the confidence term from the Criminisi priority function. Using this function essentially makes the algorithm fill patches from the outside of the hole and work its way inward.

8.3 PriorityRandom

This priority function selects a random target node to fill next. It is probably best to only use this ordering for debugging.

9 Patch Difference Functions

Patch comparisons can be done between corresponding pixels or using non-pixel specific metrics. Several patch difference functions are provided.

9.1 ImagePatchDifference

This is the "standard" patch difference function that computes a sum of differences of corresponding pixels in two patches. It is templated on the comparison to be performed on each pair of corresponding pixels.

9.2 GMHDifference

This function computes the difference between the gradient magnitude histogram of the valid region of the target patch and the gradient magnitude histogram of the entire source patch.

10 Pixel Difference Functions

The ImagePatchDifference class described above requires a function to compute the difference between pixels. Several such distance functions are provided, and the most notable ones are described here.

10.1 SumSquaredPixelDifference (SSD)

This function computes the sum of squared differences between every pixel in the valid region of the target patch and its corresponding pixel in the source patch. This is the standard difference function used in patch-based inpainting (e.g. [1]). This function is generic for any pixel type that has an operator[], but is specialized for pixels of type itk::CovariantVector. We also provide an explicitly unrolled version of this function, as since it is at the heart of the algorithm and the computational bottleneck, we have tried to do everything possible to ensure it runs as fast as possible.

10.2 SumAbsolutePixelDifference (SAD)

This function computes the sum of absolute differences between ND pixels. This function is generic for any pixel type that has an operator[], but is specialized for pixels of type itk::CovariantVector. We also provide an explicitly unrolled version of this function, as since it is at the heart of the algorithm and the computational bottleneck, we have tried to do everything possible to ensure it runs as fast as possible.

10.3 WeightedSumSquaredPixelDifference

This function computes a weighted sum of squared differences between ND pixels.

10.4 HSVSSD

The standard SSD function is acceptable if the image is represented in a color space (like RGB) where each channel is "non-wrapping". That is, the values in the upper range of the channel (255) should be significantly different from the values in the lower range of the channel (0). This is not the case with the H channel of the HSV color space, so we must treat its cyclic nature specially. In this class, we treat the S and V channels as "standard" channels, and use a special difference functor for the H channel that takes into account that we are measuring an angular distance and that wrapping over the upper range (1) back into the lower range (0) does not indicate an enormous difference, but rather we handle it correctly.

11 Drivers

As you will have noticed, the code is very heavily templated. A substantial amount of code is required to setup the objects to pass to the algorithm. To prevent code duplication when wanting to use the same algorithm in two contexts (for example, we may want a version of ClassicalImpageInpainting that displays its progress as it goes along and another that does not), we have separated this setup functionality into what we call a *driver*. This allows us to separate the data preparation from this algorithm setup. For example, We have root/ClassicalImageInpainting.cpp and root/ClassicalImageInpaintingBasicViewer.cpp both of which use the ClassicalImpageInpainting driver.

12 Interactivity

We have drivers for each of our algorithms that are called by executables with matching names that are purely terminal programs. That is, they require no GUI context to be created (they do not display anything) and do not require any GUI input from the user (they do not ask questions using QDialog, etc). In these cases, the algorithm (i.e. InpaintingAlgorithm) can be invoked as a normal function call.

However, in the case that we want to either display the intermediate progress (using BasicViewer) or prompt the user to do something like select a patch (for example, TopPatchesDialog), we must run the algorithm in a different thread so that the GUI stays responsive during the algorithm. As an example, refer to ClassicalImageInpaintingBasicViewer. Here the last line calls InpaintingAlgorithm, but is wrapped in a QtConcurrent::run call (which also requires boost::bind since the InpaintingAlgorithm is a function template). This starts the algorithm in a separate thread, and returns from the driver function to hit the app.exec() at the end of main() which triggers the GUI thread loop to start. Because of this behavior, and objects allocated on the stack (locals) in the driver function will immediately go out of scope, and nothing will work (everything will crash because the objects are no longer valid).

13 Implementation Details

13.1 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.

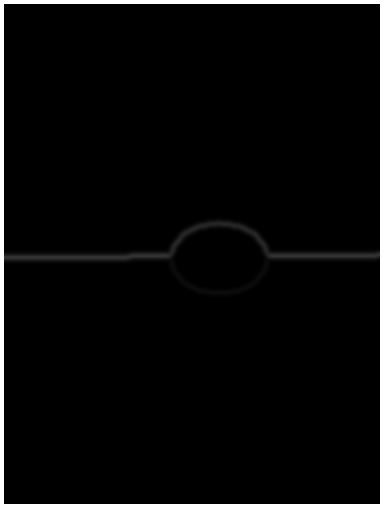
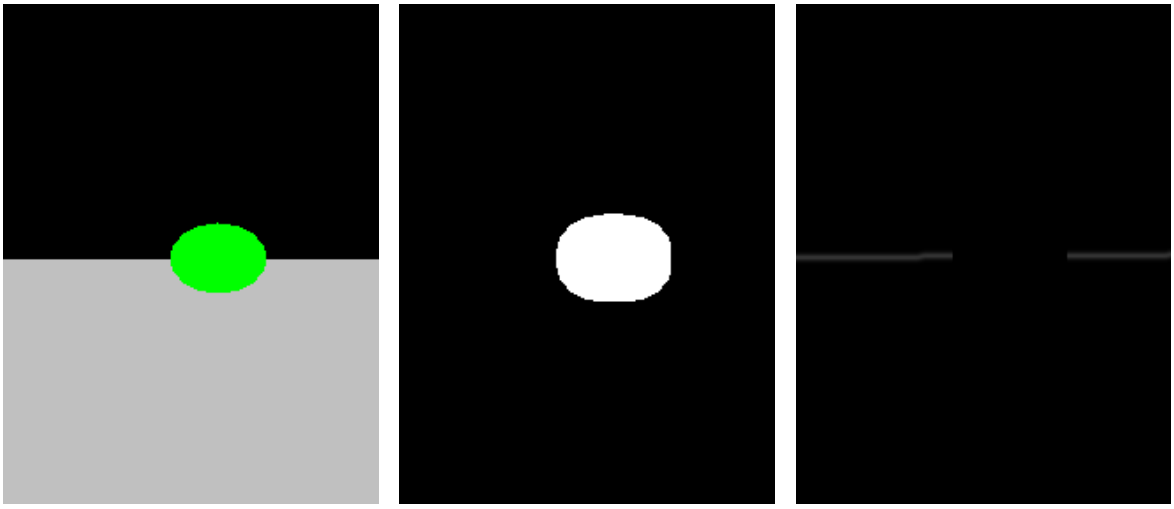


Figure 3: Result of naively computing the image gradient.

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.

13.2 Boundary Normals

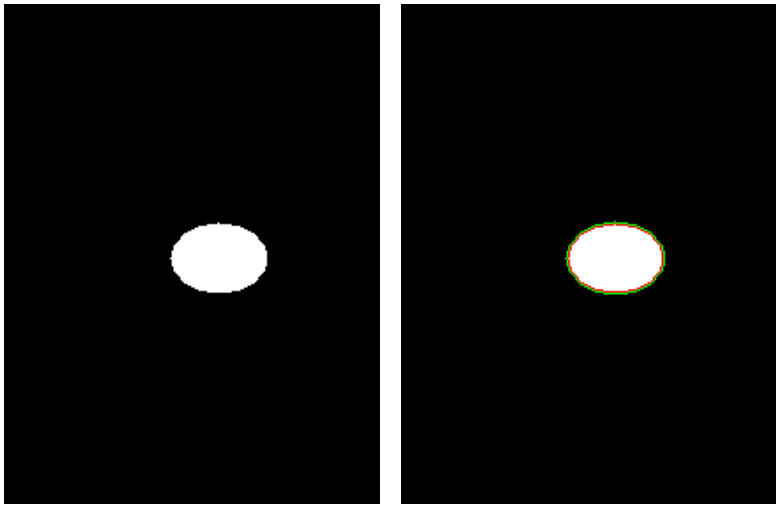
There are two things to be careful with when computing the boundary normals: computing the normals only on the one pixel thick boundary, and using the correct side of the masked region as the boundary.

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

References

[1] A. Criminisi, P. Perez, K. Toyama, *Object Removal by Exemplar-Based Inpainting*. Computer Vision and Pattern Recognition 2003 8.1, 10.1