CSC320 — Introduction to Visual Computing, Winter 2019

Assignment 2: Image Inpainting

Posted: Tuesday, February 5, 2019
Due: 12:00pm, Friday, March 1, 2019
Late policy: 15% marks deduction per 24hrs, submission not accepted if > 5 days late

---

In this assignment you will implement and experiment with an image inpainting tool. The tool is based on the *Exemplar-Based Image Inpainting* technique by Criminisi *el al.* and will be discussed in tutorials this week. Your specific task is to complete the technique's implementation in the starter code. The starter code is based on OpenCV and the Kivy user interface design library.

Goals: The goals of the assignment are to (1) get you familiar with reading and understanding a research paper and (partially) implementing the technique it describes; (2) learn how to implement basic operations such as computing image gradients and curve normals; (3) learn how to assess your implementation's correctness and the overall technique's failure points; and (4) get familiar with the event-based programming model used by typical user interfaces.

Important: You are advised to start *immediately* by reading the paper (see below). The next step is to run the reference solution as well as the starter code, and compare the differences in how they behave (*e.g.,* their choice of patches for each iteration). As in Assignment 1, you only need to understand a relatively small part of the code you are given. Once you "get the hang of it," the programming component of the assignment should not be too hard as there is relatively little python coding to do. What will take most of the time is internalizing exactly what you have to do, and how.

Testing your implementation on CDF: Unfortunately, there are some restrictions when using Kivy. The library relies on OpenGL and graphics cards and imposes some restrictions on what computers can be used, and how they can be used. Specifically, you will *not* be able to run a kivy-based executable remotely via ssh. We have also found that some of the CDF computers will give you an error because of the configuration of their GPU cards. We noticed this problem in BA2220 and BA3200 but the executable has been successfully tested in BA2210 and BA3175. That being said, the code is fully cross-platform so if you have run it successfully on your computer it should work fine on CDF too.

Starter code & the reference solution

Use the following sequence of commands on CDF to unpack and run the starter code:

```
> cd ~
> tar xvfz inpainting.tar.gz
> rm inpainting.tar.gz
> cd CS320/A2/code
> python viscomp-gui.py -- --usegui
```

Consult the file *320/A2/code/README_1st.txt* for details on how the code is structured and for guidelines about how to navigate it. In addition to the starter code, I am providing a reference solution in compiled, statically-linked binary format. The reference solution is available for OSX (*CS320/A2/code/viscomp-gui.osx*, UI not implemented in this version, but you can step through to see how the algorithm behaves) and for CDF/Linux (will be uploaded in a separate tarfile *A2-reference.cdf.tgz*). You should run the binary to see how your own implementation should behave,

and to make sure that your implementation produces the correct output. That being said, you should not expect your implementation to produce *exactly* the same output as the reference solution as tiny differences in implementation might lead to slightly different results. This is not a concern, however, and the TAs will be looking at your code as well as its output to make sure what you are doing is reasonable.

Important: For the CDF executable to run successfully, you need to copy the directory *A2/code/kv* from your starter code to the unpacked directory *viscomp-gui.cdf/*.

For those of you who do not have access to OS X, or who are not currently at a CDF computer, the file A2-referece_results.tar.gz contains screenshots of what the reference implementation produces at various stages of its execution.

320/A2/CHECKLIST.txt: Please read this form carefully. It includes information on the course's Academic Honesty Policy and contains details the distribution of marks in the assignment. You will need to complete this form prior to submission, and will be the first file markers look at when grading your assignment.

Part A: Kivy-Based User Interface (15 Marks)

Part A.1 The Run button (5 Marks)

The GUI is supposed to have a clickable 'Run' button at its lower-left corner, that is currently black. Clicking on that button should run the algorithm. If the method is not successful (*e.g.,* because one of its input images is missing) a popup window should be displayed with an error message. You need to extend the starter code to add this button to the GUI, and make sure it has the correct functionality. In particular, your GUI's behavior after pressing the button should be identical to the one of the reference implementation. To do this, you will need to complete the Kivy description file (`kivy/viscomp.kv`) and the file controlling the Kivy widgets (`inpaintingui/widgets.py`).

Part A.2 The Crosshairs (10 Marks)

After loading an image and clicking on it with the left mouse button, a red crosshair should appear, along with some text that shows the image coordinates of the pixel that was clicked. In the reference implementation, the crosshairs disappear after the button is released but in your implementation they never get erased. You need extend the starter code (`kivy/viscomp.kv` and `inpaintingui/viewer.py`) to ensure the crosshairs are erased correctly.

Part B: Exemplar-Based Image Inpainting (85 Marks)

The technique is described in full detail in the following paper (included with your starter code and also available here):

A. Criminisi, P. Pérez and K. Toyama, "Region Filling and Object Removal by Exemplar-Based Image Inpainting," *IEEE Transactions on Image Processing*, vol. 13, no. 9, 2004.

You should read Sections I and II of the paper right away to get a general idea of the principles behind the method. Section II, in particular, is very important because it introduces the notation used in the rest of the paper as well as the starter code. Section III describes the algorithm in detail, with pseudocode shown in Table I. The starter code implements exactly what is shown in Table I; the only thing left for you to implement is the term $D(\mathbf{p})$ in Eq. (1). Sections IV and V are not

strictly necessary to read, but they do show many results that should give you more insight into how your implementation is supposed to behave.

Part B.1. Programming Component (70 Marks)

You need to complete the implementation of three functions detailed below. A skeleton of all three is included in file *320/A2/code/inpainting/compute.py*. This file is where your entire implementation will reside.

In addition to these functions, you will need to copy a few lines of code from your A1 implementation that are not provided in the starter code. This requires no effort other than verbatim line-by-line copy from your existing code. See *320/A2/code/README_1st.txt* for details.

Part B.1.1. Computing Gradients: The *computeGradient()* function (30 Marks)

This function takes three input arguments: (1) a patch $\Psi_{\mathbf{p}}$ from the image being inpainted, represented as a member of the class *PSI* from file *code/inpainting/psi.py*; (2) a binary OpenCV image F indicating which pixels have already been filled; and (3) the color OpenCV image I being inpainted. The function returns the gradient with the largest magnitude within patch $\Psi_{\mathbf{p}}$:

$$computeGradient(\Psi_{\mathbf{p}}, F, I) := \nabla \tilde{I}_{\mathbf{q}^*} \quad (1)$$

$$\text{where } \mathbf{q}^* = \arg \max_{\substack{\mathbf{q} \in \Psi_{\mathbf{p}} \\ F(\mathbf{q})>0 \\ \nabla \tilde{I}_{\mathbf{q}} \text{ is valid}}} |\nabla \tilde{I}_{\mathbf{q}}| \quad (2)$$

the possible qs'
- q is a pixel that has already been filled
- q is not out of boundary of image (valid != 0)

and all gradients are computed on the grayscale version, $\tilde{I}$, of color image I.

If image I was a regular image with no "missing" pixels, implementing this function would be trivial with OpenCV: you would just need the OpenCV function that converts color images (or image patches) to grayscale, and the OpenCV function that computes the horizontal and vertical components of the gradient.

The complication here is that not all pixels $\mathbf{q}$ in $\Psi_{\mathbf{p}}$ have been filled. As a result, applying those OpenCV functions will produce estimates of $\nabla \tilde{I}_{\mathbf{q}}$ that are incorrect/invalid for some pixels $\mathbf{q}$ in that patch. Your main task in implementing *computeGradient()* will therefore be to find a way to ignore those pixels so that the max operation in Eq. (2) is not corrupted by these invalid estimates.

assume using 3x3 sobel filter … do conv on fill matrix F, where filter is all 1s', set pixel to 1 if result of conv = 255*3*3

Efficiency considerations: You should pay attention to the efficiency of the code you write but you will not be penalized for correct, yet inefficient, solutions. *Hint:* The reference implementation is 10-12 lines of code and includes no explicit looping over pixels.

Part B.1.2. Computing Curve Normals: The *computeNormal()* function (30 Marks)

Much like the previous function, this function also takes three input arguments and returns a 2D vector. The function's arguments are (1) a patch $\Psi_{\mathbf{p}}$ from the image being inpainted; (2) a binary OpenCV image F indicating which pixels have already been filled; and (3) a binary OpenCV image that is non-zero only for the pixels on the fill front $\delta\Omega$. The function assumes that the patche's center $\mathbf{p}$ is on the fill front and returns the fill front's normal $\mathbf{n}_{\mathbf{p}}$ at that pixel.

You are free to use whatever method you wish for estimating the curve normal. This includes using finite differences between $\mathbf{p}$ and its immediate neighbors on the fill front to estimate the tangent and

normal at $\mathbf{p}$; fitting a curve to the fill front in the neighborhood of $\mathbf{p}$ and returning its normal at $\mathbf{p}$; and using any built-in OpenCV functions you wish for parts of these computations.

Hint: Depending on how it is implemented this function may involve a couple dozen lines of code, but can potentially be much less (and, of course, much more).

Part B.1.3. Computing Pixel Confidences: The *computeC()* function (10 Marks)

This function takes three input arguments and returns a scalar. The function's arguments are (1) a patch $\Psi_{\mathbf{p}}$ from the image being inpainted; (2) a binary OpenCV image F indicating which pixels have already been filled; and (3) an OpenCV image that records the confidence of all pixels in the inpainted image I. It returns the value of function $C(\mathbf{p})$ shown on page 4 of the paper, *i.e.*, it assumes that the confidence of unfilled pixels is zero and returns average confidence of all pixels in patch $\Psi_{\mathbf{p}}$.

Hint: The reference implementation is less than 5 lines of code and includes no explicit looping over pixels.

Part B.2.1. Experimental evaluation (5 Marks)

Your task here is to put the inpainting method to the test by conducting your own experiments. Specifically, you need to do the following:

1. Run your inpainting implementation for 100 iterations on the test image pairs *(input-color.jpg, input-alpha.bmp)*, and *(Kanizsa-triangle-tiny.png, Kanizsa-triangle-mask-tiny.png)*. Save the partial results to directory *320/A2/report* using the following file naming convention: *X.inpainted.png, X.fillFront.png, X.confidence.png, X.filled.png* where X is either *input* or *Kanizsa*. Put these 8 images in the pdf report (see part 2.2).
2. Capture two photos, *Source1* and *Source2*, with your own camera. Any camera will do—cellphone, point-and-shoot, action camera, etc. Be sure to reduce their size to something on the order of 300×200 pixels or less so that execution time is manageable.
3. Create binary masks, *Mask1* and *Mask2*, to mask out one or more elements in these photos. Use any tool you wish for this task.
4. **Important:** Your source photos and masks should *not* be arbitrary. You should choose them as follows: (a) the region(s) to be deleted should not be just constant-intensity regions, which are trivial to inpaint; (b) the pair *Source1,Mask1* should correspond to a "good" case for inpainting, *i.e.,* should be possible to find a patch radius that produces an inpainted image with (almost) no obvious seams or other visible artifacts; and (c) the pair *Source2,Mask2* should correspond to a "bad" case for inpainting, *i.e.,* it should not be possible to find a patch radius that produces an inpainted image free of very obvious artifacts.
5. Run the inpainting algorithm on these two input datasets. You are welcome to use the reference implementation for these experiments (*i.e.,* you don't need to have your code fully functional in order to get started on this part of the assignment). Save the two sources, masks and inpainting results to directory *320/A2/report* using the file names *Source1.png, Mask1.png, Source1.inpainted.png* and *Source2.png, Mask2.png, Source2.inpainted.png*. Put these 6 images in the pdf report (see part 2.2).

Part B.2.2. Your PDF report (10 Marks)

Your report should include the following: (0) the 14 images from Part 2.1 with thier file names; (1)

why you think the pair *(Source1,Mask1)* represents a good case for inpainting; (2) why you think the pair *(Source2,Mask2)* represents a bad case for inpainting; (3) discussion of any visible artifacts in the results from these two datasets (*i.e.,* what artifacts do you see and why you think they occured).

Place your report in file *320/A2/report/report.pdf*. You may use any word processing tool to create it (Word, LaTeX, Powerpoint, html, *etc.*) but the report you turn in must be in *PDF format*.

What to turn in: Use the following sequence of commands on CDF to pack all png files in 320/A2/report and submit:

```
CHECKLIST.txt
report.pdf
algorithm.py
compute.py
viewer.py
widgets.py
viscomp.kv
```

Working on non-CDF machines: You are welcome to work on the assignment on a non-CDF machine. That being said, your implementation must run on the Linux CDF machines in order to be marked by the TAs. You should therefore test your code on CDF frequently, to make sure it has no CDF-specific issues.

All required packages have already been installed on CDF/Linux. Here are the steps I followed to install them on OS X 10.10+ :

1. Install NumPy and related packages for scientific computations:
   http://stronginference.com/ScipySuperpack/
2. Install OpenCV 3.1:
   http://blogs.wcode.org/2014/10/howto-install-build-and-use-opencv-macosx-10-10/
3. Install Kivy using homebrew and pip (needed for Part B only):
   https://kivy.org/docs/installation/installation-osx.html

Freely-available resources on NumPy, OpenCV, Computer Vision Programming, etc:

1. Short NumPy tutorial by Olessia Karpova (CSC420, 2014):
   https://github.com/olessia/tutorials/blob/master/numpy_tutorial.ipynb
2. Jan Erik Solem, *Programming Computer Vision with Python*, O'Reilly Media, 2012 (preprint):
   http://programmingcomputervision.com/ (look at Chapters 1 and 10)
3. *OpenCV-Python documentation* (Intro to OpenCV, Core Operations, Image Processing in OpenCV):
   http://docs.opencv.org/3.1.0/d6/d00/tutorial_py_root.html#gsc.tab=0
4. *Matplotlib User Guide* (especially Chapter 5.5):
   http://matplotlib.org/1.5.1/Matplotlib.pdf
5. *NumPy Reference Guide* (especially Chapters 1.4, 1.5, 3.17.1-3.17.5):
   http://docs.scipy.org/doc/numpy/numpy-ref-1.10.1.pdf