B657 Assignment 4: Stereo

Spring 2016
Due: Friday April 29, 11:59PM

In this assignment you'll work with stereo, from two different perspectives: rendering stereo from depth maps, and creating depth maps both from single images and from stereo pairs.

We've once again assigned you to a group of other students. You should only submit one copy of the assignment for your team, through GitHub, as described below. After the assignment, we will collect anonymous information from your teammates about everyone's contributions to the project. In general, however, all people on the team will receive the same grade on the assignment. Please read the instructions below carefully; we cannot accept any submissions that do follow the instructions given here. Most importantly: please start early, and ask questions on the OnCourse Forum so that others can benefit from the answers.

We once again recommend using C/C++ for this assignment, and continue to recommend using CImg and have prepared skeleton code to get you started. For this project, you *may* use the methods of the CImg class, instead of having to write everything from scratch. You may also use additional libraries for routines not related to image processing (e.g. data structures, sorting algorithms, etc.) or to what we expect for you to implement in the assignment below. Please ask if you have questions about this policy.

While you may decide to do your code development on any system you choose, **make sure your code compiles and runs on the SOIC Linux systems** because that is where we will grade them.

Academic integrity. You and your teammates may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials in the documentation of your source code. However, the code that you and your teammate submit must be your own work, which you personally designed and wrote. You may not share written code with any other students except your own teammates, nor may you possess code written by another student who is not your teammates, either in whole or in part, regardless of format.

Part 0: Getting started

Clone the GitHub repository:

`git clone https://github.iu.edu/cs-b657/`*your-repo-name*`-a4`

where *your-repo-name* is the one you found on the GitHub website.

Part 1: Creating stereograms

In class we will distribute glasses having a red lens (to be worn on your left eye, which is your right eye from the perspective of someone looking at you) and a cyan lens (for your right eye). These are a simple form of 3d viewing technology, and the mechanism is straightforward: since cyan and red are complementary colors, pixels in the red channel will not be (very) visible to the right eye, and vice-versa. To create the illusion of 3d, we can create special images that encode two separate images, each of which is visible to one eye.

Figure 1 shows a simple way of encoding a 3d image. The left image of the figure is what the image looked like from one of the two cameras that captured the stereogram. The other is a disparity map, telling you how far apart that pixel was across the two cameras. Recall that disparity and depth are inversely related.

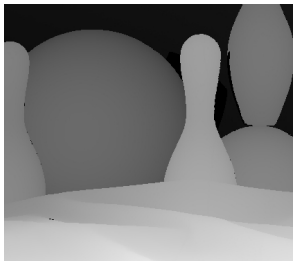


Figure 1: An image and corresponding disparity map, where brightness is inversely proportional to depth.

Write a program that accepts an image and a disparity map, like this:

```
./render image.png disp.png
```

and generates an image that appears to be in 3d when viewed through the colored glasses. We've included a few test image pairs to get you started.

Part 2: Background-foreground segmentation

What if we'd like to display an image in 3d, but only have a single image (not a pair) and no depth map? This is an active area of research. As a starting point, let's say you want to create a stereo image that has just two depths: a foreground object or region, and a background region. To do this, of course, we'll first have to segment the image into the two regions.

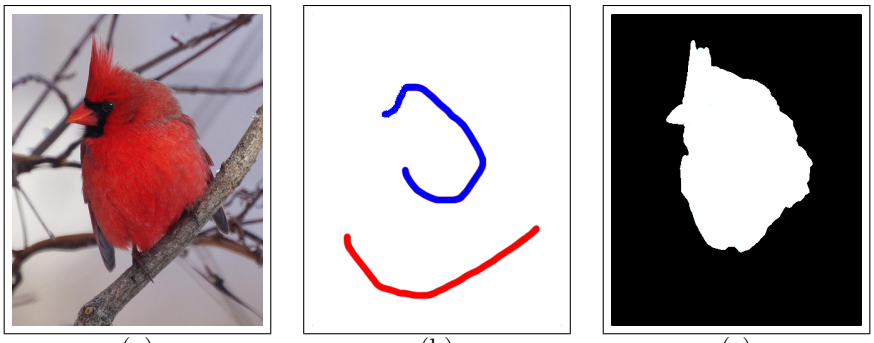
Let's use a semi-supervised approach. Our program will take a color image and a set of "seed points" that indicate where some of the foreground and background pixels in the image are located. These seed points could be quick strokes drawn manually by a human, like in Figure 2. The program will then use these seeds in an MRF model to produce a complete segmentation of the image, partitioning the image into foreground and background (Figure 2).

To get you started, we've created a skeleton program called "segment," which takes an image to be segmented and a second image with seed points. The seed points image is just a blank image with a few red and blue areas, that show the background and foreground seed points, respectively. Run the program like this:

```
./segment input.png seeds.png
```

The program should create an image that's a disparity map, that has only two possible values: some relatively high disparity for foreground pixels that should be seen as close to the viewer, and some relatively low disparity (probably 0) for background that should be far away. The existing skeleton code doesn't do very much – it just computes a random depth map.

1. As a first step towards a better segmentation algorithm, we should at least make use of the seed points provided to the program. The seed points are a set $\mathcal{F} \in I$ of pixels of image I known to be foreground points, and a set $\mathcal{B} \in I$ of points known to be background points. Based on the set of points in $\mathcal{F}$, we can build a simple appearance model of what foreground pixels "look like," for instance by simply estimating means $\mu = [\mu_R \ \ \mu_G \ \ \mu_B]^T$ and variances $\Sigma = \text{diag}(\sigma_R, \sigma_G, \sigma_B)$ of the RGB color values in $\mathcal{F}$. Let $L(p)$ denote a binary label assigned to pixel p. Then we can define a function that measures



(a) (b) (c)

Figure 2: Sample semi-supervised segmentation problem and results: (a) input image, (b) foreground (blue) and background (red) strokes drawn by a human, indicating roughly where the foreground and background regions of the image are, (c) sample disparity map produced by segmentation.

the "cost" of giving a label $L(p)$ to pixel p,

$$D(L(p), I(p)) = \begin{cases} 0 & \text{if } L(p) = 0 \text{ and } p \in \mathcal{B} \\ 0 & \text{if } L(p) = 1 \text{ and } p \in \mathcal{F} \\ \infty & \text{if } L(p) = 0 \text{ and } p \in \mathcal{F} \\ \infty & \text{if } L(p) = 1 \text{ and } p \in \mathcal{B} \\ -\log N(I(p); \mu, \Sigma) & \text{if } L(p) = 1 \text{ and } p \notin \mathcal{F} \text{ and } p \notin \mathcal{B} \\ \beta & \text{if } L(p) = 0 \text{ and } p \notin \mathcal{F} \text{ and } p \notin \mathcal{B}, \end{cases}$$

where β is a constant and $N(I(p); \mu, \Sigma)$ is a Gaussian probability density function evaluated with the RGB color values at $I(p)$. Intuitively, this says that one pays no cost for assigning a 1 to a pixel in $\mathcal{F}$ or a 0 to a pixel in $\mathcal{B}$, but we pay a very high cost for assigning the wrong label to one of these known pixels. For a pixel not in $\mathcal{B}$ or $\mathcal{F}$, we pay some fixed cost β to assign it to background, and a cost for assigning it to foreground that depends on how similar its color is to the known foreground pixels.

Implement a function called `naive_segment()` that chooses a most-likely pixel for each pixel given only the above energy function, i.e. computes the labeling,

$$L* = \arg\min_L \sum_{p \in I} D(L(p), I(p)).$$

To help you test your code, we've given you some sample images and seed files in the images/ directory of the skeleton code archive.

2. The above formulation has a major disadvantage, of course: it doesn't enforce any sort of spatial coherence on the segmentation result. To fix this, let's define a more complicated energy function,

$$E(L, I) = \sum_{p \in I} D(L(p), I(p)) + \alpha \sum_{p \in I} \sum_{q \in \mathcal{N}(p)} V(L(p), L(q))$$

where α is a constant, $\mathcal{N}(p)$ is the set of 4-neighbors of p (i.e. $\mathcal{N}(p) = \{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$ for $p = (i, j)$ not on the image boundary), and $V(\cdot, \cdot)$ is a pairwise cost function that penalizes disagreement, e.g. $V(a, b) = 0$ if $a = b$ and 1 otherwise.

Now to actually perform segmentation, we simply need to minimize the energy; i.e. find $L*$ such that,

$$L* = \arg\min_L E(L, I).$$

As we saw in class, this is an MRF energy minimization problem. Implement loopy belief propagation to (approximately) minimize this energy function. Your final program should produce two disparity maps, one for the simple approach of step 1, and one for the MRF of step 2.

3. Run your code from Part 1 on a disparity map generated by this part for a few of the sample images. How well does it work?

Part 3: Inferring depth from stereo

Now let's say you do have a stereo pair of images and would like to create a disparity map. Recall from class that, to make things easier for the stereo algorithm, image pairs are typically rectified so that the epipolar lines are parallel and correspond to horizontal scan lines of the images. We'll assume this has been done already, so you don't need to worry about it.

As we saw in class, stereo can also be posed as an MRF inference problem. (In fact, if you wrote your code in Part 2 abstractly enough, you might be able to use it directly on this problem!) Given left and right images I_L and I_R, we can define a stereo energy function that turns out to be nearly identical to that of semi-supervised segmentation,

$$F(L, I) = \sum_{p \in I} D_2(L(p), I_L, I_R, p) + \alpha \sum_{p \in I} \sum_{q \in \mathcal{N}(p)} V_2(L(p), L(q)),$$

except that we'll define the cost function D_2 differently, and the set of possible labels is no longer binary but instead the set of possible disparity values (i.e. an integer greater than or equal to 0 and less than the width of the image). For the pairwise function, one could use a variety of function as we saw in class including quadratic, but let's stick with the Potts model (the same one as in Part 2).

For the unary cost, we need a function that uses image data to infer the disparities — i.e. for each pixel (i, j) in the left image, estimates the likelihood that it corresponds to pixel $(i, j + d)$ in the right image, for all possible values of d. As we saw in class, a reasonable way of doing this is to compute sum-squared differences. The cost function then becomes,

$$D_2(d, I_L, I_R, (i, j)) = \sum_{u=-W}^{u=W} \sum_{v=-W}^{v=W} (I_L(i + u, j + v) - I_R(i + u, j + v + d))^2.$$

Complete `stereo` by implementing loopy belief propagation to minimize the energy function F. To help you test your code, we've included a few stereo pairs in your repo. The skeleton code can compute an error measure (the mean squared difference) between the disparity map you produce and the ground truth. The skeleton code also already implements a "naive" version of the MRF inference which uses D_2 but assumes $V_2(L(p), L(q)) = 0$; this simplifies the problem dramatically, but should at least help you get started. What is the best quantitative error with respect to ground truth that you can achieve? Qualitatively, how do the stereograms produced by Part 1 look when fed with your inferred disparity map?

What to turn in

Make sure to prepare (1) your source code (**including the test images**), and (2) a report that explains how your code works, including **the exact command to run each subproblem**, any problems you faced, and any assumptions, simplifications, and design decisions you made, and answers **ALL** the questions posed above. To submit, simply put the finished version (of the code and the report) on GitHub (remember to `add`, `commit`, `push`) — we'll grade the version that's there at 11:59PM on the due date.