MIT EECS 6.815/6.865: Assignment 4:

HDR and Tonemapping

Due Wednesday October 14 at 9pm

1 Summary

- merge N bracketed images into an HDR image

- tone mapping with Gaussian and Bilateral blurs

2 HDR merging

Important. To merge images to HDR, we will work with images encoded linearly. Most of the digital images you may encounter are encoded with a gamma of 2.2 (in particular those of the handout). Therefore, in this problem set, you will have to undo the gamma 2.2 from your inputs before merging them to HDR. You will also have to apply a gamma 2.2 to the images you output in the linear domain (e.g. after tone-mapping). We provide you with a function changeGamma to do this. Follow the example in the main file.

In the whole assignment, we will only work with images of static subjects and the camera hasn't moved between shots in the same sequence.

Consult the course slides for the overall HDR merging approach. We will first compute weights for each pixel and each channel to eliminate values that are too high or too low. We will then compute the scale factor between the values of two images, to determine their relative exposures.

Finally, we will merge a sequence of images using a weighted combination of the individual images according leveraging the weights and scale factors.

1.a **Weights.** Write a function Image computeWeight(const Image &im, float epsilonMini=0.002, float epsilonMaxi=0.99) that returns an image with pixel value 1.0 when the corresponding pixel value of im is between epsilonMini and epsilonMaxi, and 0.0 otherwise. The weights are computed on a per pixel and per channel basis.

1.b **Factor.** Now that we know which pixels are usable, we can compute the multiplication factor between a pair of images. Write a function float computeFactor that takes two images and their weights computed using the above method, and returns a *single* scalar corresponding to the median value of im2/im1 for pixels that are usable in

---

> both `im1` and `im2`. Add an epsilon of 10^{-10} to the pixels of im1 to avoid divide by 0 errors.

With these two methods, we know which pixels are usable in each image and what the relative exposures between two images are, we can merge a sequence of images to a single High-Dynamic Range image.

For the rest of the problem, you can assume that the first image in the sequence is the darkest, and that images are given in order of increasing exposure.

Do not forget the special case seen in class to compute the weight of the darkest and brightest images. You should only threshold in one direction for these cases. If a pixel is not assigned to any of the weight images, then assign it the corresponding value from the first image in the sequence (by doing this you should avoid any divide by 0 errors as well). To test your method, you can write out the output image scaled by different factors.

As you merge pixels do not forget to use scale them with $factor$.

`testMakeHDR` in `a4_main.cpp` illustrates the process for the `design` image sequence.

---

2 **Merge to HDR** Write a function
`Image makeHDR(const vector<Image> imageList,`
`float epsilonMini=0.002, float epsilonMaxi=0.99)` that takes
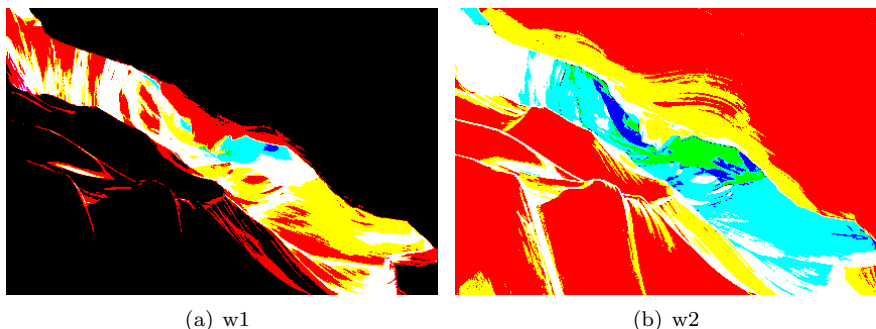a sequence of images as input and returns a single HDR image.

(a) w1   (b) w2

Figure 1: The weights computed from the `ante2` image sequence using the parameters provided in `testComputeFactor`

3 Tone mapping

We have assembled our first HDR image, but this image still cannot be displayed properly on our low dynamic range screen. Let's implement tone mapping to remap the HDR information to a displayable range.

Your tone mapper will follow the method studied in class. The function is called `toneMap`. It takes as input an HDR image, a target contrast for the base

(lowpass) layer, an amplification factor for the detail, and a Boolean to switch the lowpass/highpass separation between the bilateral filter or Gaussian blur.

As described in the lecture, our goal is to reduce the contrast from the HDR image (say, 1:10000) to what the display can show (say 1:100). Although gamma correction would be the first thing we can think of, this results in washed out images, as shown on the slide "Naïve: Gamma compression". The colors are actually okay (they're all there) but the high frequencies are washed out. We therefore want to work on the luminance, and increase the high frequencies, and we want to work in the log-domain, since the human eye is very sensitive to multiplicative contrast (recall lecture 2)!

We want to modify only on the log-luminance:

3.a In the function `toneMap`, first decompose your image into luminance and chrominance using the function from problem set 1 `std::vector<Image> lumiChromi(const Image &im)` that can be found in `basicImageManipulation.cpp`. We also provide you the reciprocal function `lumiChromi2rgb`.

3.b Next, compute a log10 version of the luminance. Add a small constant (e.g. the minimum non-zero value) to the luminance to avoid divergence at 0.

We recommend you do this by first implementing the helper function `float image_minnonzero(const Image &im)`, followed by `Image log10Image(const Image &im)`, for which we have written function signatures and comments for you. Then, call `log10Image` with the luminance image as the argument.

Next, we want to extract the detail of the (log) luminance channel. We do this by blurring the log luminance, and subtracting the original. If you recall problem set 2, this gives you the the details (high frequencies)!

4.a You are ready to compute the *base (blurry) luminance*. We will implement two versions: Gaussian blur and bilateral filtering [a], which will be chosen based on the value of the input parameter `useBila`. In both cases, we will use a standard deviation for the spatial Gaussian equal to the biggest dimension of the image divided by 50. The parameter `truncateDomain` should be set to the default value of 3.

You are welcome to use your own implementation of filtering methods, but you can also use our versions in `filtering.cpp`.

4.b Given the base, compute the detail by taking the difference from the original log luminance.

---
[a] recall why the bilateral filter is important here: see "The halo nightmare" class slide.

Great, you are finally ready to put everything together! Make sure you understand the slides *Contrast reduction in log domain* that combine the base

luminance, detail and brightness scaling factor!

> 5.a Compute the scale factor `k` in the log domain that brings the dynamic range of the base layer to the given target (that is, the range in the log domain should be `log10(targetBase)` after applying `k`. We have provided two new functions for you that you might find useful: `float Image::min() const` and `float Image::max() const`. Get a new base luminance using the factor, and multiply the details (in the log domain) by `detailAmp`, then add the resulting output base and detail. Make sure to add an offset that ensures that the largest base value will be mapped to 1 once the image is put back into the linear domain
>
> 5.b Convert this new luminance back to the linear domain (you may want to separately implement `Image exp10Image(const Image &im)`). Then, reintegrate the chrominance into the resulting image. We've provided the function `Image lumiChromi2rgb(const Image &lumi, const Image &chromi)` in `basicImageManipulation.cpp` which might be helpful.

Enjoy your results and compare the bilateral version with the Gaussian one. Use the functions `testToneMapping_ante2`, `testToneMapping_ante3, testToneMapping_design, testToneMapping_boston` in `a4_main.cpp` to help test your tone mapping function.

Note: The bilaterial filter on the `design` image sequence takes a very long time. It is not necessary to test bilaterial filter tone mapping for this image sequence.
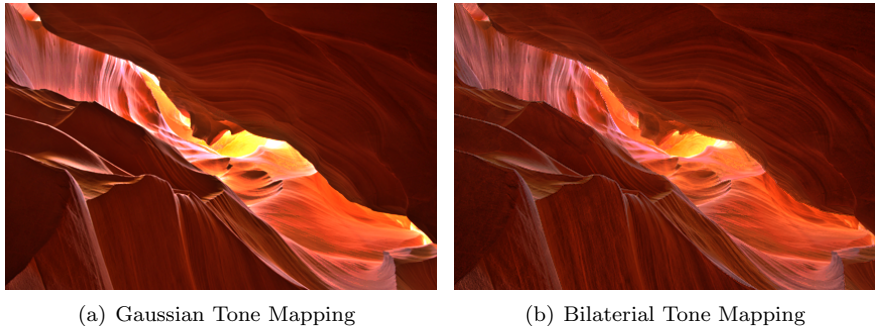

(a) Gaussian Tone Mapping   (b) Bilaterial Tone Mapping

Figure 2: Tone mapping of the `ante2` image sequence using the parameters provided in `testToneMapping_ante2`

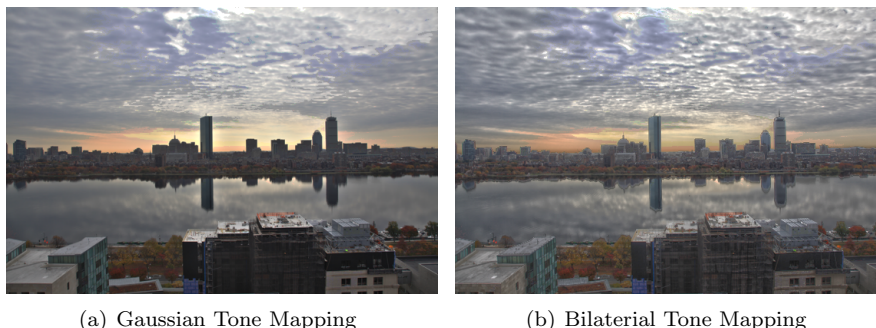(a) Gaussian Tone Mapping   (b) Bilaterial Tone Mapping

Figure 3: Tone mapping of the `boston` image sequence using the parameters provided in `testToneMapping_boston`

4   Extra credit (10% max)

- (5%) Deal with image alignment (e.g. the `sea` images. We recommend Ward's median method, but probably single-scale to make life easier yet slower. Alternatively, you can simulate the clipping in the darker of two images.

- (5%) Derive better weights by taking noise into account. You can focus on photon noise alone. This should give you an estimate of standard deviation (or something proportional to the standard deviation) for each pixel value in each image. Use a formula for the optimal combination as a function of variance to derive your weights. This should replace the thresolding for dark pixels, but you still need to set the weight to zero for pixels dangerously close to 1.0.

- (5%) Write a function to calibrate the response curve of a camera. See `http://www.pauldebevec.com/Research/HDR/`

- (10%) Implement the bilaterial grid for fast bilateral filtering. See `http://groups.csail.mit.edu/graphics/bilagrid/` with more mathematical justifications at `http://people.csail.mit.edu/sparis/publi/2009/ijcv/Paris_09_Fast_Approximation.pdf`

- (10%) Hard: deal with moving objects.

5 Submission

Turn in your files to the online submission system (link is on Stellar) and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)

- Potential issues with your solution and explanation of partial completion (for partial credit)

- Any extra credit you may have implemented and their function signatures if applicable

- Collaboration acknowledgment (you must write your own code)

- What was most unclear/difficult?

- What was most exciting?