Homework Assignment 1

15-463/663/862, Computational Photography, Fall 2021
Carnegie Mellon University

Due Friday, Sep. 17, at 11:59pm ET

This assignment has two parts. The purpose of the first part is to introduce you to Python as a tool for manipulating images. For this, you will build your own version of a very basic image processing pipeline (without denoising). You will use this to turn the RAW image into an image that can be displayed on a computer monitor or printed on paper. The Python packages required for this assignment are `numpy`, `scipy`, `skimage`, and `matplotlib`. The purpose of the second part is to introduce you to very basic principles of image formation and exposure control. For this, you will build your own simple *camera obscura*, which is basically a fancy term for a pinhole camera.

There is a "Hints and Information" section at the end of this document that is likely to help. We strongly recommend that you read that section in full before starting to work on the assignment.

1 Developing RAW images

For this problem, you will use the file `campus.NEF` included in the `./data` directory of the assignment ZIP archive. This is a RAW image that was captured with one of the course DSLR cameras (Nikon D3400). As we discussed in class, RAW images do not look like standard images before first undergoing a "development" process[1]. The developed image should look *something* like the image in Figure 1. The final result can vary greatly, depending on the choices you make in your implementation of the image processing pipeline.

Figure 1: One possible developed version of the RAW image provided with the assignment.

[1] The term "develop" comes from the analogy with the process of developing film into a photograph. As discussed in class, the same process is often called "rendering" the RAW images

1.1 Implement a basic image processing pipeline (80 points)

RAW image conversion (5 points). The RAW image file cannot be read directly by skimage. You will first need to convert it into a .tiff file. You can do this conversion using a command-line tool called dcraw (https://www.dechifro.org/dcraw/). After you have downloaded and installed dcraw, you will first do a "reconnaissance run" to extract some information about the RAW image. For this, call dcraw as follows:

```
dcraw -4 -d -v -T <RAW_filename>
```

In the output, you will see (among other information) the following:

```
Scaling with darkness <black>, saturation <white>, and
multipliers <r_scale> <g_scale> <b_scale> <g_scale>
```

Make sure to record the integer numbers for <black> and <white>.

Calling dcraw as above will produce a .tiff file. Do *not* use this! Instead, delete the file, and call dcraw once more as follows (note the different flags):

```
dcraw -4 -D -T <RAW_filename>
```

This will produce a new .tiff file that you can use for the rest of this problem.

Python initials (5 points). We will be using skimage function imread for reading images. Originally, it will be in the form of a numpy 2D-array of unsigned integers. Check and report how many bits per pixel the image has, its width, and its height. Then, convert the image into a double-precision array. (See numpy functions shape, dtype and astype.)

Linearization (5 points). The 2D-array is not yet a linear image. As we discussed in class, it is possible that it has an offset due to dark noise, and saturated pixels due to over-exposure. Additionally, even though the original data-type of the image was 16 bits, only 14 of those have meaningful information, meaning that the maximum possible value for pixels is 4095 (that's $2^{12} - 1$). For the provided image file, you can assume the following: All pixels with a value lower than <black> correspond to pixels that would be black, were it not for noise. All pixels with a value above <white> are saturated. The values <black> for the black level and <white> for saturation are those you recorded earlier from the reconnaissance run of dcraw.

Convert the image into a linear array within the range $[0, 1]$. Do this by applying a linear transformation (shift and scale) to the image, so that the value <black> is mapped to 0, and the value <white> is mapped to 1. Then, clip negative values to 0, and values greater than 1 to 1. (See numpy function clip.)

Identifying the correct Bayer pattern (20 points). As we discussed in class, most cameras use the Bayer pattern in order to capture color. The same is true for the camera used to capture our RAW image.

We do not know, however, the exact shift of the Bayer pattern. If you look at the top-left 2×2 square of the image file, it can correspond to any of four possible red-green-blue patterns, as shown in Figure 2.

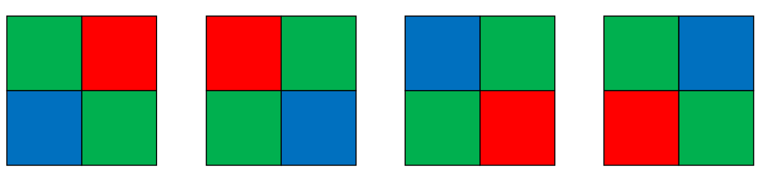

Figure 2: From left to right: 'grbg', 'rggb', 'bggr', 'gbrg'.

Think of a way for identifying which Bayer pattern applies to your image file, and report the one you identified. It will likely be easier to identify the correct Bayer pattern after performing white balancing.

White balancing (10 points). After identifying the correct Bayer pattern, we want to perform white balancing. Implement both the white world and gray world automatic white balancing algorithms, as discussed in class. Additionally, implement a third white balancing algorithm, where you multiply the red, green, and blue channels with the <r_scale>, <g_scale>, and <b_scale> values you recorded earlier from the reconnaissance run of `dcraw`. These values correspond to the white balancing presets used by the camera. *After completing the entire developing process*, check what the image looks like when using each of the three white balancing algorithms, and decide which one you like best. (See `numpy` functions `max` and `mean`.)

Demosaicing (10 points). Once white balancing is done, it is time to demosaic the image. Use bilinear interpolation for demosaicing, as discussed in class. Do not implement bilinear interpolation on your own! Instead, use `scipy`'s built-in `interp2d` function.

Color space correction (10 points). You now have an RGB image that is viewable with the standard `matplotlib` display functions. However, the image color does not look quite right. This is because the image pixels have RGB coordinates in the color space determined by the camera's spectral sensitivity functions, and not in the "standard" color space that image viewing software expects. We use scary quotes for "standard" because what color space the image viewing software *actually* assumes, and what color space it *should* assume, are neither the same nor trivial to determine. We will discuss these issues during the color lecture, but in the meantime, a good default choice is to use the *linear sRGB color space* [1].

To transform your image to the linear sRGB color space, implement a function that applies the following linear transformation to the RGB vector $[R_{\text{cam}}, G_{\text{cam}}, B_{\text{cam}}]^\top$ of *each* pixel of your demosaiced image:

$$\begin{bmatrix} R_{\text{sRGB}} \\ G_{\text{sRGB}} \\ B_{\text{sRGB}} \end{bmatrix} = \left(\mathbf{M}_{\text{sRGB}\to\text{cam}}\right)^{-1} \cdot \begin{bmatrix} R_{\text{cam}} \\ G_{\text{cam}} \\ B_{\text{cam}} \end{bmatrix}. \tag{1}$$

The 3×3 matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ transforms a 3×1 color vector from the sRGB color space to the camera-specific RGB color space. We use its inverse to apply the transformation in the reverse way.

Computing the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ involves a sequence of steps that will make a lot more sense once we have gone through the color lecture later in the course. Here, we describe these steps at a high level, with just enough detail for you to be able to implement them. We write the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ as:

$$\mathbf{M}_{\text{sRGB}\to\text{cam}} = \mathbf{M}_{\text{XYZ}\to\text{cam}} \cdot \mathbf{M}_{\text{sRGB}\to\text{XYZ}}. \tag{2}$$

As the notation suggests, the 3×3 matrix $\mathbf{M}_{\text{sRGB}\to\text{XYZ}}$ transforms a 3×1 color vector from the sRGB to the XYZ color space; and analogously, the 3×3 matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ transforms a 3×1 color vector from the XYZ to the camera-specific RGB color space. Their product, then, implements the transformation from the sRGB to the camera-specific RGB color space, as we want. The XYZ color space is an intermediate *reference* color space that color management systems use to accurately perform color space transformations.

The *sRGB standard* [1] provides the following values that you can use in your implementation:

$$\mathbf{M}_{\text{sRGB}\to\text{XYZ}} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix}. \tag{3}$$

You can find the values of the camera-specific matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ in the raw code of `dcraw`, under the function `adobe_coeff`. Look for the entry corresponding to the camera used to capture the RAW image you are developing. Note that the `dcraw` code provides the matrix values multiplied by $10,000$, so make sure to adjust the matrix you use in your implementation accordingly. Additionally, the `dcraw` code shows the matrix as a 1×9 vector, which you should rearrange to a 3×3 matrix assuming row-major ordering.

After computing the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ as in Equation (2), normalize it so that its rows sum to 1. You need to do this so that you do not have to perform white balancing a second time. Finally, plug the normalized matrix into Equation (1) to correct the color space of your image.

Brightness adjustment and gamma encoding (10 points). You now have a 16-bit, full-resolution, linear RGB image. Because of the scaling you did at the start of the assignment, the image pixel values may not be in a range appropriate for display. Moreover, the image is not yet gamma-encoded. As we discussed in class, this means that when you display the image, it will appear very dark.

Brighten the image by linearly scaling it. Select the scale to set the post-brightening mean grayscale intensity to some value in the range $[0, 1]$. After the scaling, clip all intensity values greater than 1 to 1. (See skimage function rgb2gray for converting the image to grayscale, and numpy function clip for clipping.) What the best value is is a highly subjective judgment, so you should experiment with many percentages and report and use the one that looks best to you. For the photographically inclined, selecting a post-brightening mean value of 0.25 is equivalent to scaling the image so that there are two stops of bright area detail.

You are now one step away from having an image that can be properly displayed. The last thing you need to take care of is tone reproduction (also known as gamma encoding). For this, implement the following non-linear transformation, then apply it to the image:

$$C_{\text{non-linear}} = \begin{cases} 12.92 \cdot C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308, \\ (1 + 0.055) \cdot C_{\text{linear}}^{\frac{1}{2.4}} - 0.055, & C_{\text{linear}} > 0.0031308, \end{cases} \tag{4}$$

where $C = \{R, G, B\}$ is each of the red, green, and blue channels. This odd-looking function is not arbitrary: it corresponds to the tone reproduction curve specified in the sRGB standard [1], which also specifies the sRGB color space you used earlier. It is a good default choice if the camera's true tone reproduction curve is unknown. We will discuss sRGB during the color lecture, but you are welcome to read up on it on Wikipedia.

Compression (5 points). Finally, it is time to store the image, either with or without compression. Use skimage function imsave to store the image in .PNG format (no compression), and also in .JPEG format with the quality parameter set to 95. This setting determines the amount of compression. Can you tell the difference between the two files? The compression ratio is the ratio between the size of the uncompressed file (in bytes) and the size of the compressed file (in bytes). What is the compression ratio?

By changing the JPEG quality settings, determine the lowest setting for which the compressed image is indistinguishable from the original. What is the compression ratio?

1.2 Perform manual white balancing (10 points)

As we discussed in class, one way to do manual white balancing is by: 1) selecting some patch in the scene that you expect to be white; and 2) normalizing all three channels using weights that make the red, green, and blue channel values of this patch be equal. Implement this algorithm, and experiment with different patches in the scene. Show the corresponding results, and discuss which patches work best. (See matplotlib function ginput for interactively recording image coordinates).

1.3 Learn to use dcraw (10 points)

Besides converting RAW files to .tiff, dcraw provides options to emulate all steps in the image processing pipeline, including steps that are camera-dependent. Read through dcraw's documentation, and figure out what the correct flags are in order for dcraw to perform all the image processing pipeline steps you implemented in Python. Make sure to report the exact dcraw flags you used.

Note that dcraw outputs images in .PPM format and does not have an option to use the .PNG format. You will need to use an image editor to convert the image format. The command-line utility pnmtopng, available in the Netpbm toolkit (http://netpbm.sourceforge.net/), is an easy way to do the conversion. Alternatively, ImageMagick (https://www.imagemagick.org/) also does command-line format conversions.

You should now have (at least) two developed versions of the original RAW image: One (or more) you produced using your implementation of the image processing pipeline, and another you produced using dcraw. One more developed image is available as the file campus.jpg included in the ./data directory of the assignment ZIP archive. This is the developed image produced by the camera's own image processing pipeline, alongside the RAW image you have been using. Compare the three developed images, and explain any differences between them. Which of the three images do you like best?

2 Camera Obscura

A camera obscura is essentially a dark box with a pinhole on one face (hence also "pinhole camera"), and a screen on the opposite face. Light reflecting off an object travels through the pinhole to the screen, and forms an inverted image of the object on the screen. All it takes to make such a camera is having a paper box and something you can use to pinch small holes on it. The caveat is that it will be hard to see the image formed with your naked eye. Instead, you will need to attach to the pinhole camera a digital camera with a long exposure time. Figure 3 shows a schematic of a pinhole camera constructed in the way described above.

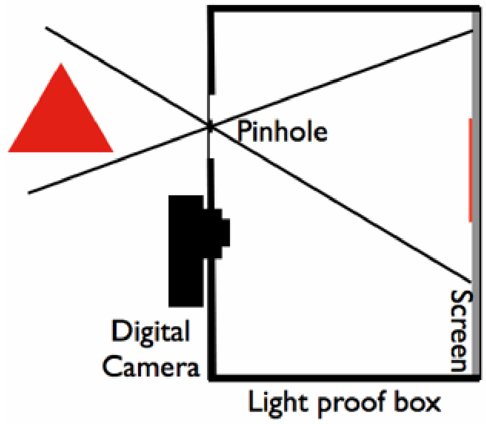

Figure 3: Schematic of a camera obscura.

2.1 Build the pinhole camera (70 points)

To design a basic easy-to-use pinhole camera, we recommend the following steps:

- Get a cardboard box. It does not have to be too big, but it does have to ultimately be "lightproof". A shoebox will do. The cardboard box should be such that the distance between the pinhole and the screen (i.e., the focal length) is longer than the minimum focus of the digital camera. Otherwise, all the images you will capture will be blurry.
- Obtain a digital camera that allows for long exposure times, around 15-30 seconds.
- Determine which face of the box should be your screen. Cover the screen with white paper (printer paper generally works fine). Cover the rest of the faces on the inside with black paper. If you prefer, you can use white and black paint instead of sheets of paper.
- On the face opposite the screen, cut a large hole. This hole should be bigger than the pinhole, e.g., about 1 inch in diameter.
- Take another piece of black paper, and punch the pinhole into that. Then, tape this piece of paper over the bigger hole on the box, so that light only enters through the pinhole. The advantage of this is that you can use multiple pieces of paper to test pinholes of varying diameters. Make your design so that you can easily switch between papers with different pinhole diameters.
- Next to the hole for the pinhole-carrying paper, cut a hole for the digital camera. The size of the hole should be such that you can attach the digital camera's aperture to it in a light-tight way—no light should enter through this hole. Additionally, the digital camera's hole should not be too far from the pinhole, as otherwise the digital camera's field of view may not be wide enough to capture the screen's image. You may need to angle the digital camera a bit towards the pinhole. At the same time, you should make sure that the digital camera is not blocking the light path inside or outside the box.
- Make sure that your digital camera has a memory card and a charged battery. Then attach it to the box. While attaching it, make sure that it is still possible to change its settings (e.g., focus, exposure, ISO) without destroying all of your construction. Turn off autofocus and adjust the focus of the camera manually so that it is focused on the screen.

- Duct tape your box all over! You really want to make it lightproof.

Once you have completed everything, submit a few photos of your constructed pinhole camera. Make sure to also report on all your design decisions, such as screen size, focal length, and field of view.

2.2 Use your pinhole camera (30 points)

It is now time to capture some images with your pinhole camera. We leave it up to you to identify interesting scenes. Once you have found what you want to photograph, point the pinhole towards it. Figure out the appropriate settings for the digital camera, then set it to capture an image for 16-30 seconds.

You should capture at least three scenes, each with three different pinhole diameters, for a total of nine photographs. Some suggested pinhole diameters are 0.1 mm (really just a pinprick), 1 mm , and 5 mm. These diameters are suggestions: in reality, your pinhole diameter should be about $1.9\sqrt{(f\lambda)}$, where f is the focal length, and λ is the wavelength of light (550 nm on average, for visible light). If you use this formula, then also go a few millimeters up and down, in order to have three pinhole diameters in total.

Report what pinhole diameters you use, and discuss the differences you observe for the different pinholes.

2.3 Bonus: Camera obscura in your room (20 points)

Instead of using a cardboard box, you can build a camera obscura using a room with a window. Use thick black paper to cover the window, and pinch a small hole at its center. Then, on the wall opposite the window, you will see an inverted projection of the view outside your room. Use a digital camera to capture an image showcasing this projection. See Abelardo Morell's Camera Obscura gallery for inspiration and some further instructions. Antonio Torralba and Bill Freeman also wrote a fun computer vision paper about room-sized pinhole cameras [2], which is definitely worth a read.

3 Deliverables

When submitting your solution, make sure to follow the homework submission guidelines available on the course website (`http://graphics.cs.cmu.edu/courses/15-463/assignments/submission_guidelines.pdf`). Your submitted solution should include the following:

- A PDF report explaining what you did for each problem, including answers to all questions asked throughout Problems 1 and 2, as well as any of the bonus problems you choose to do. The report should include any figures and intermediate results that you think may help. Make sure to include explanations of any issues that you may have run into that prevented you from fully solving the assignment, as this will help us determine partial credit. The report should also explain any `.PNG` files you include in your solution (see below).
- All your Python code, including code for the bonus problems you choose to complete, as well as a `README` file explaining how to use the code.
- For Problem 1.1: At least two `.PNG` files, showing the final images you created with the two different types of automatic white balancing. You can also include `.PNG` files for various experimental settings if you want (e.g., different brightness values).
- For Problem 1.2: `.PNG` files, showing the results of manual white balancing for different patch selections.
- For Problem 1.3: The `.PNG` file produced by `dcraw`.
- For Problem 2.1: At least two photographs of your constructed pinhole camera.
- For Problem 2.2: At least nine photographs captured with your pinhole camera (three different scenes, each captured with three different pinholes).
- For Bonus Problem 2.3: A photograph of your room's "screen" with the projection of the view from outside, as well as a photograph of your covered window "pinhole".

4 Hints and Information

- Make sure to download and install the latest version (version 9.28) of `dcraw`. In particular, the default version that comes in older Windows versions does not support the cameras used in this class, and will produce results with a strong purple hue.
- We recommend using Anaconda (https://www.anaconda.com/) for Python package management. Once you install `conda`, you can create an environment specifically for the class using the following command:

  ```
  conda create --name <ENV-NAME> numpy scipy scikit-image
  ```

  If you feel more comfortable using some other Python package management system, such as `pip`, you are welcome to do so. Either way, make sure you have *at least* versions 1.19.1 for `numpy`, 1.5.0 for `scipy`, 0.16.2 for `skimage`, and 3.3.1 for `matplotlib`. Newer versions should be fine.
- The package `matplotlib` is included with the `skimage` installation and is very useful for visualizing intermediate results and creating figures for the report. For example, the following code reads in two images and creates a single figure displaying them side by side:

  ```
  import matplotlib.pyplot as plt
  from skimage import io
  # read two images from current directory
  im1 = io.imread('image1.tiff')
  im2 = io.imread('image2.tiff')

  # display both images in a 1x2 grid
  fig = plt.figure()          # create a new figure
  fig.add_subplot(1, 2, 1)    # draw first image
  plt.imshow(im1)
  fig.add_subplot(1, 2, 2)    # draw second image
  plt.imshow(im2)
  plt.savefig('output.png')   # saves current figure as a PNG file
  plt.show()                  # displays figure
  ```

  Note that figures can also be saved by clicking on the save icon in the display window.

  When displaying grayscale (single-channel) images, `imshow` maps pixel values in the $[0, 1]$ range to colors from a default color map. If you want to display your image in grayscale, you should use the following instead:

  ```
  # read a single-channel image
  imgr = io.imread('image_gray.tiff')
  plt.imshow(imgr), cmap='gray')
  plt.show()
  ```

  You will need this in several places in this assignment, as many of your images will be grayscale.
- You can access subsets of `numpy` arrays using the standard Python slice syntax `i:j:k`, where `i` is the start index, `j` is the end index, and `k` is the step size. The following example, given an original image `im`, creates three other images, each with only one-fourth the pixels of the originals. The pixels of each of the corresponding sub-images are shown in Figure 4. You can also use the `numpy` function `dstack` to combine these three images into a single 3-channel RGB image.

  ```
  import numpy as np
  # create three sub-images of im as shown in figure below
  ```

```
im1 = im[0::2, 0::2]
im2 = im[0::2, 1::2]
im3 = im[1::2, 0::2]

# combine the above images into an RGB image, such that im1 is the red,
# im2 is the green, and im3 is the blue channel
im_rgb = np.dstack((im1, im2, im3));
```

Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	
Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	
Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	

- You will find it very helpful to display intermediate results while you are implementing the image processing pipeline. However, before you apply brightening and gamma encoding, you will find that displayed images will look completely black. To be able to see something more meaningful, we recommend scaling and clipping intermediate image `im_intermediate` before displaying with `matplotlib`.

```
plt.imshow(np.clip(im_intermediate*5, 0, 1), cmap='gray')
plt.show()
```

  Figure 4 shows what you should see with this command after the linearization step.

  Additionally, the colors of intermediate images will be very off (e.g., have a very strong green hue), even if you are doing everything correctly. You will not get reasonable colors until after you have performed at least white balancing. This will come into play when you are trying to determine the Bayer pattern.

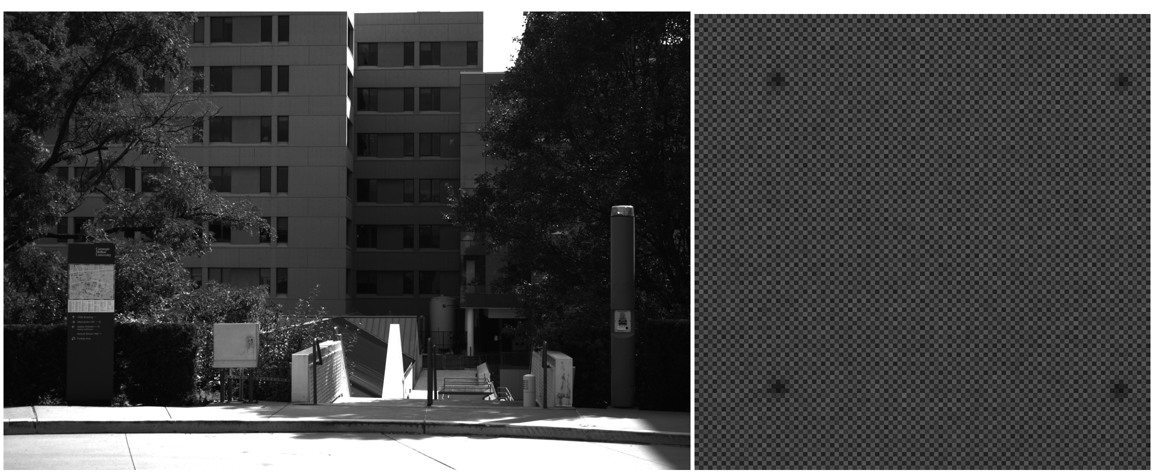

Figure 4: Left: The linear RAW image (brightness increased by 5). Right: Crop showing the Bayer pattern.

- If you decide to build your camera obscura using the camera provided by the course (recommended), make sure to go over the camera tutorial available on the course website (http://graphics.cs.cmu.edu/courses/15-463/index_files/camera_tutorial.pdf).

- In many point-and-shoot cameras, as well as in most manual-control cameras (DSLR, rangefinder, mirrorless, and so on), exposures up to 30 seconds are readily available. If this exceeds your camera's settings, you can reduce exposure times by using a large aperture size for your lens, as well as a large ISO setting (possibly up to 1600).

  Modern smartphone cameras typically have a hardware limit on maximum exposure time at around 4 seconds. However, many phones nowadays support a "long exposure" mode, where longer exposure times are simulated by capturing a sequence of images and summing them all up at the RAW stage. This article provides some information for both iOS and Android phones.

  If you decide to use your smartphone, you should also use an app that enables manual control of camera settings (especially focus, exposure time, and ISO). Additionally, make sure to disable the flash.

- Below are some tips that will hopefully make capturing images with your pinhole camera easier:

  1. We strongly recommend that you go outside during a sunny day. Capturing images indoors will require extremely long exposures, which is impractical.
  2. Make sure to place your pinhole camera on a surface that is absolutely steady throughout the exposure time. Definitely do not hole it with your hand.
  3. When focusing your digital camera, do so with the aperture of your lens fully open (even if you later decide to stop down the aperture while capturing images.
  4. If your photographs are blurry, it could be because either your camera is not focused correctly on the screen, or the pinhole is too large. To make sure that it is not a focusing issue, place a piece of paper with text on the screen and take a photo (or look at the viewfinder of your digital camera) with your box open. Make sure the camera is focused so that the text appears sharp. If the issue is the size of your pinhole, try another one that is smaller.

5 Credits

A lot of inspiration for the first part of this assignment came from Robert Sumner's popular guide for reading and processing RAW files. Some components of the first part were adapted from James Tompkin's computational photography course at Brown. Some inspiration for the second part came from James Hays' and Gordon Wetzstein's computational photography courses, at Brown and Stanford respectively.

References

[1] International Electrotechnical Commission and others. IEC 61966-2-1:1999. *Multimedia systems and equipment–Colour measurements and management–Part 2-1: Colour management–Default RGB colour space–sRGB*, 1999.

[2] A. Torralba and W. T. Freeman. Accidental pinhole and pinspeck cameras: Revealing the scene outside the picture. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 374–381. IEEE, 2012.