Image compositing for photographic lighting

Tjaart van der Walt

May 5, 2015

Contents

1 Introduction

This project is an implementation of the concepts found in the paper User-assisted Image Compositing for Photographic Lighting[1] by Ivaylo Boyadzhiev, Sylvain Paris and Kavita Bala. All the mathematical models found in this document is taken from the paper. The contribution of this project is that it provides a Free Software implementation of these concepts.

We want to provide better software support for a new workflow that has been emerging in the area of photography of static scenes. This workflow differs from the traditional workflow in the fact that it does not require the photographer to meticulously consider his lighting setup before taking a photo. With the new workflow the photographer takes multiple photos of the scene from a fixed vantage point[2] with a moving light source. Photographers currently using this workflow then exports these images into an image manipulation software package and manually edits the input images into one well lit output image.

This project provides support to automate a lot of this work that can currently only be done by image manipulation experts. We provide 6 command line scripts[3] to attain this goal.

Light type	Description
Fill light	Provides even illumination across the image
Edge light	Emphasizes the main edges in the scene
Diffuse color light	Emphasizes the base colors of objects
Per object modifier	Emphasizes a particular object
Soft light modifier	Removes harsh shadows and highlights
Regional lighting modifier	Balances the overall lighting of the scene

[1] http://www.cs.cornell.edu/projects/light_compositing/
[2] Typically a tripod.
[3] For detailed usage see the Usage section.

2 Installation

2.1 Source code

The source for the package is available on GitHub[4]

2.2 Installation

There are two ways to easily install the `light-compositing` package. The first is to use the Python `Distutils` directly. For this you will have to download the source and execute the `setup.py` script in the root directory using the command:

```
python setup.py install
```

The second way is to install it from the `Python Package Index`. We choose to use `pip` [5] as our packaging tool, as this seems to be the standard for Python going forward. The advantage of doing it this way is that your Python dependencies can be automatically resolved by the packaging tool[6].

```
pip install light-compositing
```

Both of these installation methods installs the `light_compositing` package into the Python `site-packages` [7] directory. On a Unix based system it will also install the executable scripts into the `/usr/bin` directory.

2.3 Python notes

2.3.1 Python 2 vs. Python 3

Our system is developed for Python 2. Some of the frameworks we used are not yet Python 3 compatible, and so we chose to work with the older version. Theoretically, if you could install Python 3 compatible versions of all the dependencies, this program should work with Python 3.

2.3.2 Pip

`pip` is a command line tool that allows you to install packages from the `Python Package Index`. The tool handles all dependency management, making it a very appealing way to install Python packages.

If you have Python 2.7.9 or later installed, the `pip` command is included with the interpreter. Otherwise you can install `pip` using this[8] script.

[4] https://github.com/tjaartvdwalt/light-compositing

[5] If you don't have Python version 2.7.9 or above installed, see the notes on how to install `pip` in the section below.

[6] The `PPI` will use the same `setup.py` script as the manual installation, but will automatically resolve dependencies like NumPy and and PyIpopt

[7] Located at `/usr/lib/python2.7/site-packages/light_compositing-0.0.0-py2.7.egg` on my system.

[8] https://bootstrap.pypa.io/get-pip.py

```
python get-pip.py
```

2.4 Dependencies

2.4.1 Python interpreter

You will need a working Python 2 interpreter. The minimum working version is unknown at this time, but any 2.7 series interpreter should work fine.

You can download the Python interpreter from here.[9]

On Arch Linux you can install Python using the package manager.

```
pacman -S python2
```

Our testing environment used Python 2.7.9.

2.4.2 OpenCv

You will need a working OpenCv2 installation, with the Python bindings installed.

You can download OpenCv from here.[10]

On Arch Linux you can install OpenCv using the package manager.

```
pacman -S opencv
```

Our testing environment used OpenCv 2.4.10

2.4.3 NumPy

In addition to the Python interpreter, you will need to have the NumPy library installed. The easiest way to get this is by using `pip`

```
pip install numpy
```

2.4.4 Scipy

We use the `scipy.optimize` package to solve our minimization problems. This package supports various optimization techniques. Initially we chose the `Nelder-Mead` method, since this was the easiest to implement, but we found it very slow to converge on a solution.

Eventually we settled on the Truncated Newton method `TNC`. This method does a lot of heavy lifting for us by estimating the gradient numerically.[11] This means that we can converge on a solution with much fewer iterations. We are also able to provide constraints for our optimized values. For example we use the constraints to stop the algorithm from finding negative coefficients, which would be meaningless for us.

`SciPy` can be installed from `PyPI` using the following command:

```
pip install scipy
```

[9] https://www.python.org/downloads/

[10] http://opencv.org/downloads.html

[11] This is exactly what the paper mentions should be done.

3 Test data

We have included our test data in the `test_data` sub directory of our submission. This is a subset of the images used by the authors of the paper that this implementation is based on. We do not own the copyright of these images, but they appear to be in public domain. The original images can be downloaded from here. [12]

3.1 Input images

We chose two sets of input images from the available test data to do our development.

3.1.1 Basket

The basket of fruit gives us the stereotypical use case for a static image. The basket is contains several distinct objects providing a good baseline for our edge light model. The shadows cast around the base of the bowl also provide a good test for soft lighting modifier. This set consists of 128 images. In figure 1 we see a few representative images from this set.

Figure 1: Representative images from the basket of fruit set. This set consists of 128 images in total

3.1.2 Cafe

In contrast to the basket image set, the cafe provides a much larger scene, so that a much smaller area of the scene is lit in any one image. This is a good test for the fill light model, because the input light will be less uniform than with the basket images. It also gives us the best opportunity to test our regional lighting modifier. The red chair in the cafe is used in the paper as the baseline for diffuse color light. This image set consists of 113 images. In figure 2 we see a few representative images from the set.

[12] http://www.cs.cornell.edu/projects/light_compositing/download/light_compositing_input_data.zip

Figure 2: Representative images from the cafe set. This set consists of 113 images in total

4 Usage

Currently our application does not contain a fully fledged graphical user interface. Each light type is a script that can be run from the command line.

4.1 Command line options and arguments

Most of the scripts takes a `directory` argument. This is the directory where the input images are found. We assume that these images are named in the format `000.png ... xyz.png`. If no `--count` option is given, these image files should be the only images in the directory. [13]

Most of the scripts have a `--count` option, that allows you to set how many of the input images should be used for the calculation.[14].This is useful if you do not want to waste time doing calculations on all the input images. From the results in the paper it seems that setting `COUNT > 15` will yield good results in most cases.

All of the scripts have a `--verbose` option. Setting this option prints debug information.

All of the scripts have a `--output` option. You can use this option to set the name of the output file you want to generate. If left unset a file name will be based on the light type that was used to generate it. For example running the `fill_light` script will produce `fill_light.png`

The edge light, and diffuse color light scripts have an option called `--downsample`. The option value determines how many times the input images will get down sampled before being passed to the optimization algorithm. The resulting lambdas are then applied to the original sized images. This is useful if you want to get an approximate solution in quick time.

4.2 Basis lights

Depending on the size and number of input images used it can take a long time to compute the basis lights. It is best to do this ahead of time, and then use

[13] We use the number of files in the directory as the default value for `COUNT`.

[14] Unfortunately, this setting always uses the first `count` images in the directory, so you cannot rerun the program with a set of randomly selected input images at this stage.

the results as input for the light modifiers. The basis lights only have to be calculated once per set of input images.

4.2.1 Fill light

```
fill_light test_data/basket
```

4.2.2 Edge light

```
edge_light test_data/basket
```

4.2.3 Diffuse color light

```
diffuse_color_light test_data/basket
```

4.3 Light modifiers

The light modifiers can be calculated in real time. With further development these scripts could be incorporated into a graphical user interface that allows a user to manipulate the lighting of his image.

4.3.1 Per-Object modifier

The per object modifier takes the output files generated by the `fill_light`, `edge_light` and `diffuse_color_light` scripts as arguments. If these arguments are not specified it uses the default values for each of these lighting types.

The per object modifier has a `--mask` option that gives the path to a binary image that defines the object of interest. In addition it has `--fill` `--edge` and `--diffuse` options that gives the weight of each lighting type for this modifier.

In this example we give each light type equal weight.

```
object_modifier --mask test_data/basket_regions/corn.png \
>  --fill 1 --edge 1 --diffuse 1 fill_light.png \
>  edge_light.png diffuse_color_light.png
```

4.3.2 Soft light modifier

The soft light modifier has a `--sigma` option that defines the σ value. This value controls how much a particular light is diffused as described in the implementation section.

```
soft_light_modifier --sigma 0.5 test_data/basket
```

4.3.3 Regional lighting modifier

The regional light modifier takes an input image as argument instead of an input directory like most of the other lighting types. In addition it has a `--beta` option that defines the β value. When this value is positive, it emphasizes the bright areas in the input image, and when it is negative it emphasizes dark areas in the input image.

```
regional_light_modifier --beta 0 my_image.png
```

5 Implementation

5.1 Basis lights

5.1.1 Fill light

Fill light tries to give even illumination everywhere in the input image. We assume that our input lights will be approximately uniformly distributed across the input images.

Model Formally we try to minimize:

$$I_{\text{fill}}(p) = \frac{\sum_i w_i(p) I_i(p)}{\sum_i w_i(p)}$$

Where

$$w_i(p) = \frac{\overline{I}_i(p)}{\overline{I}_i(p) + \epsilon}$$

$$\overline{I}_i(p) = I_i(p) \cdot (0.2990, 0.5870, 0.1140)$$

$$\epsilon = 0.01$$

Results For our results we compare our fill light with an unweighted average of the input images. In figure we see the comparison by using all 128 of the basket images as input. We note that there is not much difference between the two images.

To illustrate the value of the weight component in our fill light, we chose 30 of the cafe images in which the right side was more prominently lit than the left side. In figure we can see that the unweighted average performs poorly, but the fill light maintains good results.

5.1.2 Edge light

Edge light tries to emphasize the main edges in a scene. We leverage the fact that the main edges of the scene will always be located in the same place in the image, whereas occasional shadows and highlights will be sporadically distributed over the input images.

Figure 3: Fill light applied to the bowl of fruit image set.
Left: Average light, **Right:** Fill light

Figure 4: Fill light applied to the cafe image set. The images were selected in such a way that the left side is left relatively dark.
Left: Average light, **Right:** Fill light

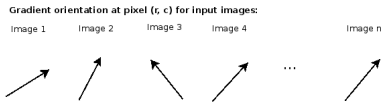
Model

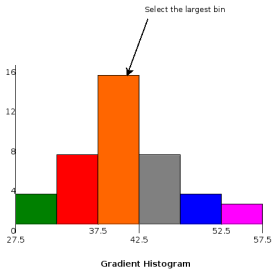
$$I_{\text{edge}} = \sum_i \lambda_i I_i$$

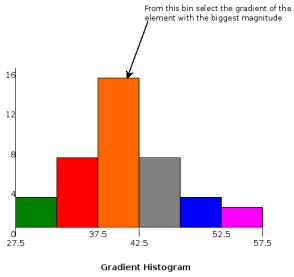
$$\arg\min_{\{\lambda_i\}} \sum_p h(p)||\nabla\left(\sum_i \lambda_i I_i(p)\right) - G(p)||^2$$

Where G is the gradient map of the combined input images. $h(p)$ is the per pixel weight designed to give more weight to pixels with a well defined orientation.

Notes We use the Sobel mask to get the gradient orientations.







We use PyIpopt. . . . . .

Results

5.1.3 Diffuse color light

Diffuse color light tries to emphasize the base color of objects.

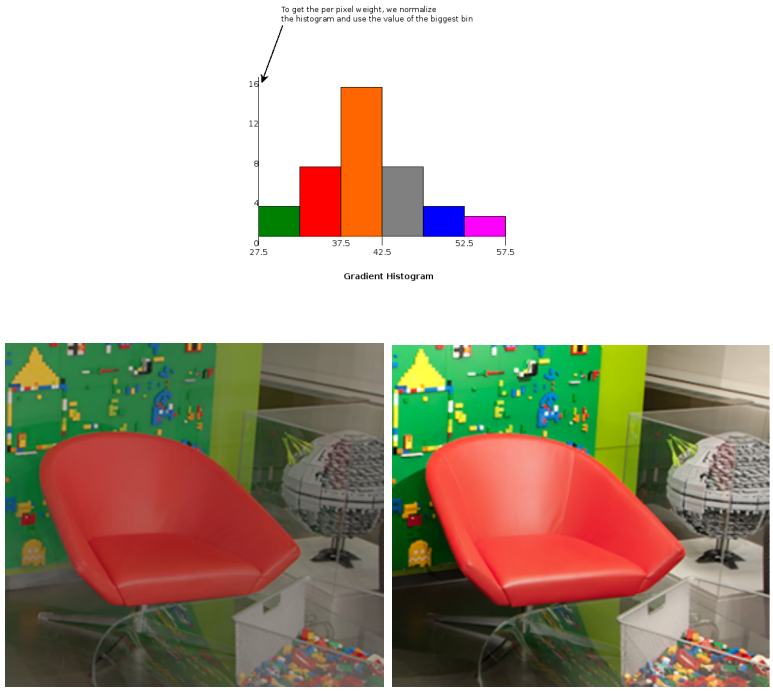


Figure 5: Edge light applied to the cafe set of images.
Left: Fill light, **Right:** Edge light

Model For this model we handle colorful objects and neutral colored objects seperately. For colorful objects we define a function that favors diffuse reflection instead of the specular reflection. A key insight is that the specular component appears white, so the stronger it is the less colorful the object appears.

$$\mathbf{I} = d\mathbf{D} + s\mathbf{W}$$

The saturation angle is given by $\angle(I, W) = \arccos\left(\frac{I}{||I||} \cdot \frac{W}{||W||}\right)$

where $W = (1, 1, 1)^T$,

We notice that for a fixed value d the angle decreases as s increases. We want to choose λ s that maximizes the angle between the pixels in our input images and the specular component.

$$\arg\max_{\{\lambda_i\}} \sum_p \hat{w}(p) \angle \left(\sum_i \lambda_i I_i(p), W \right) \tag{1}$$

Similarly to the fill light, we apply weights to discourage the selection of dark pixels.

$$\hat{w}(p) = \frac{\sum_i \lambda_i I_i(p)}{\sum_i \lambda_i I_i(p) + \epsilon}$$

For neutrally colored objects we define a function that encourages similarity between the average image and our result.

$$\arg\min_{\{\lambda_i\}} \sum_p \angle \left(\sum_i \lambda_i I_i, \frac{1}{N} \sum_i I_i \right) \tag{2}$$

To make sure we apply this only to neutral colors, we define a balancing term:

$$\alpha(p) = \exp\left(\frac{-\angle \left(\frac{1}{N} \sum_i I_i(p), W \right)^2}{2\sigma^2} \right)$$

By combining equation 1 and equation 2 we define our diffuse color light as:

$$I_{\text{diffuse}} = \sum_i \lambda_i I_i$$

where

$$\arg\min_{\{\lambda_i\}} \sum_p \left[\alpha(p) \angle \left(\sum_i \lambda_i I_i(p), \frac{1}{N} \sum_i I_i(p) \right) - (1 - \alpha(p)) \hat{w}(p) \angle \left(\sum_i \lambda_i I_i(p), W \right) \right.$$

Notes

Results

5.2 Light modifiers

5.2.1 Per-Object modifier

The per-object modifier controls the spread of light. It is inspired by photographic equipment like a snoot.

Model

- We compute the basis lights as discussed in the previous section.
- We apply them only to pixels within a selected object.
- The users can then mix these colors interactively using 3 track bars.
- To ensure smooth blending with the rest of the image we apply a bilateral filter

Figure 6: Diffuse color light applied to the cafe set of images.
Left: Fill light, **Right:** Diffuse color light

Notes For our implementation the objects will be defined by a mask image. This image contains white pixels in the object area, and black pixels in the background area. If he had created a GUI we would have allowed the user to define the object using a pointing device.

Results

5.2.2 Soft light modifier

The soft light modifier produces areas with soft shadows and highlights. It is inspired by photographic equipment like umbrellas and soft boxes

Model

$$I_{\text{soft}} = \sum_i \lambda_i I_i$$

where

$$\text{soft}_{\sigma_S}(\Lambda) = \frac{||\Lambda||}{||S\Lambda||} S\Lambda$$

$$\Lambda = (\lambda_1, \ldots, \lambda_N)^T$$

$$S_{ij} = \exp\left(\frac{-||I_i - I_j||^2}{2\sigma_s^2}\right)$$

Notes

Results

Figure 7: Soft light modifier applied to the bowl of fruit images.
Left: Fill light, **Right:** Soft light modifier

Figure 8: Soft light modifier applied to the cafe images.
Left: Fill light, **Right:** Soft light modifier

5.2.3 Regional lighting modifier

The regional lighting modifier balances the light intensities in a scene to emphasize (or de-emphasize) a specific region in a scene. It provides a way to adjust the light balance across the scene at a coarse level.

Model

- We are only interested in balancing the lighting, so we work with the intensity images $\bar{I}_i$
- We calculate the first component using Principle Component Analysis. This captures the different areas of illumination
- We translate our image to the log domain $P = \ln(\bar{I}_i)$
- We ensure the overall image intensity remains unchanged by enforcing a zero mean on P: $\hat{P} = P - \frac{1}{N}\sum_p P(p)$
- To remove undesirable boundaries from $\hat{P}$ we apply a bilateral filter
- We create a map $M = e^{(\beta \hat{P})}$
- Our modifier is defined by: $I_{\text{regional}} = \frac{1}{N}\sum_i M I_i$

We observe that if:

- $\beta = 0$ the result remains unchanged
- $\beta > 0$ emphasizes regions where $\hat{P} > 0$
- $\beta < 0$ emphasizes regions where $\hat{P} < 0$

Notes To get the intensity image $\bar{I}_i$ we convert our `BGR` image to `HSV` color space. Then we extract the `V` component and work with that as our intensity image. Once we have done our calculations we return the `V` component to our `HSV` image, and convert it back to a `BGR` image.

The easiest way that I could find to extract the `V` component using `NumPy` appears to be extracting it line for line.

```
hsv_image = cv2.cvtColor(soft_light_img, cv2.cv.CV_BGR2HSV)

# shape[0] gives the number of rows in the image
# [:, 2] gets the second component from each element int the row
for i in range(hsv_image.shape[0]):
    hsv_image[i][:, 2] = hsv_image[i][:, 2]
```

We compute the first PCA component using the built-in OpenCv function

```
mean, e = cv2.PCACompute(log_row, maxComponents=1)
```

In the paper they use a cross-bilateral filter, but for our implementaiton we simply use the built-in bilateral filter. The choice of parameters are as recommended in the OpenCv documentation.

```
filtered_pca_hat = cv2.bilateralFilter(pca_hat, 5, 50, 50)
```

Figure 9: Regional light modifier applied to the cafe fill light image.
Left: $\beta > 0$, **Right** :$\beta < 0$

Results

6 Future work

One important way in which we could improve the speed of our implementation is to make use of GPU acceleration. Ideally we would like to have a library that allows NumPy to do calculations directly on the GPU. A brief search revealed a library called Theano [15] that appears to suit our needs.

Another feature that would make our implementation more approachable to the general public would be to add a graphical user interface. Since this is a proof of concept we deemed this outside the scope of the project.

[15] http://www.deeplearning.net/software/theano/