Detection of Light Sources and Shadow

SOKVATHARA LIN 6018002,
u6018002@au.edu

MENH KEO 6038302,
u6038302@au.edu

Computer Science Student,
Assumption University,
Samut Prakan, Thailand

Abstract

The majority of tasks are associated with image processing or augmented reality include rendering new objects into existing images or matching objects with unknown illumination. To encourage such algorithms, it is crucial to conclude from which directions a scene was illuminated and whether only a photograph is provided. Equally important, we present a completely unique source of illumination detection algorithm that, contrary to this state-of-the-art, is in a position to detect multiple light sources with sufficient accuracy. 3D measures are not required, only the input image and a really bit of unskilled user interaction.

I. Introduction

One image has been created by many pixels combine. Each part of the image contains a value which calls a threshold that defines how light or dark that part is. With the help of a threshold, we can define or detect many different things in one object. Therefore, in this project, we would like to present how to detect an image's light source and shadow based on an algorithm that enhances the threshold value. After the enhancement, we can use those results to determine the shadow areas of the image.

When detecting light sources, it is essential to indicate the actual relative intensities of light sources in a scene, given only a photograph as input. Moreover, it is also a complex method to disclose and pinpoint the accuracy's light source. However, if the light source is known, it can be approximately recovered and indicate the specific positions.

Several methods compare the invariant color spaces, including HIS, HSV, HCV, YIQ, and YCbCr model for shadow detection. Nevertheless, this project will use the normalized difference vegetation index (NDVI) [3] to define the color invariant and extract shadow segments by automatic thresholding the grayscale image.

II. Light Source Detection

In this approach, the reading image from the input source using OpenCV. Furthermore, the image data that has been read, its value stores in RGB values that needed to invert R and B channels result in BGR. Moreover, the conversion needs **COLOR_HSV2BGR** from OpenCV that backward image value to RGB/BGR.

RGB ↔ HSV

In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit the 0 to 1 range.

$$V \leftarrow max(R,G,B)$$

$$S \leftarrow \begin{cases} \frac{V-min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/(V-min(R,G,B)) & \text{if } V = R \\ 120+60(B-R)/(V-min(R,G,B)) & \text{if } V = G \\ 240+60(R-G)/(V-min(R,G,B)) & \text{if } V = B \end{cases}$$

If $H < 0$ then $H \leftarrow H + 360$. On output $0 \le V \le 1, 0 \le S \le 1, 0 \le H \le 360$.

The values are then converted to the destination data type:

- 8-bit images: $V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2$(to fit to 0 to 255)
- 16-bit images: (currently not supported) $V < -65535V, S < -65535S, H < -H$
- 32-bit images: H, S, and V are left as is

Now, after operating, we will be able to access each pixel in an image. Additionally, after the conversion's operation on original image data, its result with BGR image, three values per pixel (B, G, and R), separated them conveniently. Of course, as a result, it can be shown and be viewed in Figure 1. This will lead to extract a color object in the following method.

Original BGR

Figure 1

Since there are several color-space conversion methods available in OpenCV. However, we chose COLOR_HSV2BGR because it is applicable in the light tracking of our method. The next step is, we have to smooth the image by using **cv2.GaussianBlur()** to do blur the current image. Identically, instead of a box filer consisting of equal filter coefficients, a Gaussian kernel is used. And, it is done with the function that has been mention above. Again, we have chosen the kernel (5, 5), which specific the width and height of the kernel that will be applied and result positive and odd. Additionally, the standard deviation in the X and Y directions are zeros because we only wanted to calculate the kernel size. Gaussian filtering is very effective in extracting Gaussian noise from the image **(Figure 2).** Then, you will see that the second image is smoother and more precise.

BGR Gaussian Blur

Figure 2

As in any other signal, there are different types of noises in the image because of the source (camera sensor). The techniques to do smoothing by reducing the noise is the best fundamental in image processing.

The next step is focusing on Canny Edge Detection. Canny Edge Detection is a popular edge detection algorithm. John F. Canny developed it in 1986. It is a multi-stage algorithm, and we will go through each stage [1]. Since we have removed the obstacle, which is the noise from the image with a 5x5 Gaussian filter, we can start finding the intensity gradient of the image.

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Gradient direction is always perpendicular to edges. It is rounded to four angles representing vertical, horizontal, and two diagonal directions [1].

This stage will use OpenCV's **cv2.Canny()** which has the Algorithm above to do the operation. And, the first argument is our input image. The second and third arguments are our minVal and maxVal, respectively. Again, the first threshold for the hysteresis procedure that we have chosen is 225, and the second threshold is 250. These values will be used for edge linking. Additionally, the most significant value between the first and second threshold will be the firm edges. We want to track edges by hysteresis: Finalize the detection of edges by suppressing all other edges that are weak and not connected to firm edges [2]. As shown above, the result of applying **cv2.Canny()** is shown in **(Figure 3).**

Canny Edge Detection

Figure 3

At this stage, we start finding contours in the image. Since the image has been optimized with the desired threshold to get the right edges within the image, the Canny Edge Detection has come in handy since OpenCV **findContours()** will have better accuracy with binary data of the image. Moreover, the OpenCV 3.2 or above has many convenient things that do not modify the source image. And, in OpenCV, finding contours will indicate a white object from the black background. Also, the object that has been found will be white, and the background will result in black.

In addition, there are three arguments in **cv2.findContours()** function. The first is the source image which is the binary canny edge image, and the second is contour retrieval mode which we chose **cv2.RETR_EXTERNAL** as the option, and the third is the contour approximation method that will be **cv2.CHAIN_APPROX_NONE** as the last argument. Furthermore, the values as the result **Contours** by the OpenCV might get us to be concerned with the external contours of the image, as we are looking for the individual light objects of our image and not the internal contours within a light object. Again, we have created a custom function **get_contour_areas()** which will find the areas of each of the contours within our image. And, this has been done by using the **cv2.contourArea()** function **(Figure 4).**

```
def get_contour_areas(cnts):
    all_areas = []
    for cnt in cnts:
        area = cv2.contourArea(cnt)
        all_areas.append(area)
    return all_areas
```

Figure 4

We then sort the list from largest to smallest contours and store them in the new sorted value list. Moreover, the sort that we used is the **sorted()** function with the key: **cv2.contourArea**. Also, to accomplish sort from largest to smallest, we need to set **the reverse** value to **True.**

Finally, we then loop all sorted contour lists to start outlining each contour in the original image to indicate the light source with the image (Figure 5). And the contours that have been drawn are in yellow.

Original Light Source Detection

Figure 5

A. Experiment and Result

In this section, we choose another image that contains many light bulbs with all lights turn on. Moreover, its background has many windows and shapes that might confuse the Canny Edge Detection because many objects have obvious shapes in the background. However, since we have indicated the Canny Edge Detection using "**Tight Threshold**" between 225 and 250, the edge results have become cleaner and conveniently track the desired edges. Furthermore, the complex result of light detection with complex background objects results in **Figure 6.**

Original Light Source Detection

Figure 6

B. Conclusion

In this study, we have disclosed the light detection method, and the concept requires to use OpenCV only to achieve this detection. Moreover, color-space conversions, blur images with various low pass filters, remove noises, configure the right low and high threshold, generate binary data, and sort the value list are crucial to accomplish this method.

On the other hand, if it desires to have a very high detection rate of light objects, that might need to adjust some more threshold range to indicate the specific desire detection of the light objects within the image. Again, most of the light sources within the image that we tested are acceptable, as the results have shown and met with the demand objectives.

III. Shadow Detection

In this project, we choose color invariants. Color variants is a method that helps to extract the color properties of an object. So, we follow the formula [3][4] to define a color invariant.

$$\psi r = \frac{4}{\pi} arctan(\frac{B-G}{B+G})$$

Where B stands for blue band and G stand for the green band, with this formula, we will get the dark color of the image, and since the shadow is part that seems darker than other parts, this formula will be a good and easy way to calculate the shadow ratio.

In **Figures 7** and **8**, after we apply the color invariants method to the original, we obtain another picture containing only the darker object, which we can see that it is the shadow. With the color variant method, the shadow seems darker than the other part, which is a benefit for us to detect only the shadow. We can call the result of this color invariant as shadow ratio since it shows only the shadow. From the result, we can see that the shadow is darker than the other part. We know that the shadow threshold is more significant than another part (the non-shadow part).

Original Applied Formula

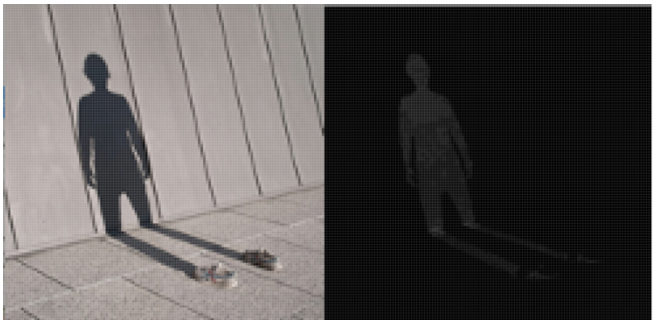

Figure 7

Original Applied Formula

Figure 8

The next step is to create a shadow mask from the threshold that contains a value greater than the mean of the shadow mask (picture that we obtain after we apply the color invariants method.). The shadow will contain only the shadow, but to make the picture smooth, we apply a kernel to make the picture clearer. After the enhancement, the picture still contains many small unnecessary objects that we should consider removing out **Figure 9**.

After extract the right threshold

Figure 9

This project uses a library function from '**skimage'** called **remove_small_objects (Figure 10)**. After we remove all the unnecessary objects, we will obtain a shadow mask that contains shadow. With this shadow mask, we can use it to create contour or boundary on the original picture base on the coordinate of the shadow mask.

Remove unnecessary small objects

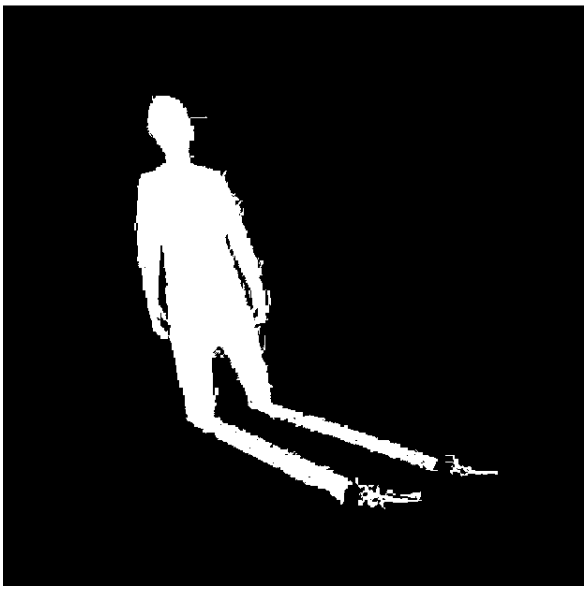

Figure 10

A. Experiment and Result

In this section, we provide two test images; one contains many shadows, but all the shadows are separate clearly from the backgrounds because the background color is distinguished with shadow, so we call this a best-case scenario. Another picture contains many objects, but some appear to be black, similar to a shadow, and we call this worst-case scenario.

In **Figure 11**, the picture in the best-case scenario shows that with color invariant, we can easily separate between a shadow and background, and it also detects the shadow with an accuracy rate 95%.

Original Shadow Detection

Figure 11

Nevertheless, in **Figure 12**, it has appeared to us that we can detect all shadows. Meanwhile, it detects some objects that are not the shadow due to darker colors that seem confused between shadow and object. Therefore, in this worst-case scenario, we can see that color invariant can separate color and detect shadow with an accuracy rate of about 75%-80%.

Original Shadow Detection

Figure 12

B. Conclusion

In this study, we present a simple shadow detection method that is easy to understand and apply the whole concept to any project by applying a color invariant formula. Color invariant will separate the color of shadow and background and create a shadow mask that we can use to detect shadow. However, this method requires a specific determination of threshold which reasons regarding a shadow or create an automatic function that compares threshold with the minimum threshold from shadow mask. On the other hand, if the requirement is to get a high detection accuracy rate, it needs to have a picture with different background colors compared to shadow. Otherwise, the result will be inaccurate if the other object appears to have a similar color to shadow.

IV. REFERENCES

[1]. JOHN F. CANNY IN 1986, "DEVELOPMENT OF THE CANNY ALGORITHM", WIKIPEDIA: 22 JULY 2020

[2]. OPENCV DOCUMENTATION, OPENCV API REFERENCE (FINDS EDGES IN AN IMAGE USING THE [CANNY86] ALGORITHM.), IMAGE PROCESSING, DOCS.OPENCV.ORG: 31 DECEMBER 2019

[3]. C. UNSALAN AND K. L. BOYER, "LINEARIZED VEGETATION INDICES BASED ON A FORMAL STATISTICAL FRAMEWORK" , IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, VOL. 42, PP 1575-1585,2002

[4]. BERIL SIRMACEK AND CEM UNSALAN, "DAMAGED BUILDING DETECTION IN AERIAL IMAGES USING SHADOW INFORMATION", IEEE XPLORE, JULY 2009