Methods For Better Color Tracking
As An Extension To The Human Paintbrush

Tyler Sammann[*]
Rensselaer Polytechnic Institute

Abstract

The Human Paintbrush is an interactive piece made for two users; one that gives instructions (the Painter), and one that receives and attempts to follow instructions (the Human Paintbrush). The blindfolded paintbrush receives instructions via sound, from eight surrounding speakers. The painter controls these sound instructions, attempting to influence the paintbrush's movements in particular ways. These movements, and other actions, are tracked, and are converted into brush-strokes on a digital painting.

The project currently uses a GigEVision color camera, and tracks the position of the paintbrush based on color (as the paintbrush wears a blue Snuggie). The current method of color tracking is very primitive and simple. A simple threshold for determining blue is hand calibrated for the current space being tracked, but small changes in the light source's color and intensity have a significant impact on the correctness of the threshold and accuracy of the result. This paper discusses different methods of transformation of the original image in order to correct this problem. RG-normalization is implemented to allow for color invariance given different intensities of illumination. Two color constancy algorithms are also implemented and compared to see which allows for the least color variation given different colors of illumination. A combination of normalization and color constancy is applied to create a more robust color tracker that is more insensitive to changes in room illumination, and requires less time for thresholding. This paper also discusses different methods for blob detection as a form of noise reduction. A bounding boxes algorithm, as well as a connected components algorithm are implemented and tested.

Keywords: color tracking, color constancy, connected components

1 Introduction

This project is based on communication between three separate programs. Both the sound and the entire interface for the painter were created with a MaxMSP patch. A C++ program using the GigEVision SDK interfaces with the camera, and using a basic color tracking method, tracks the position of the paintbrush (who wears a blue Snuggie). The MaxMSP program receives position data from this program, allowing the painter to see the location of the paintbrush, and allowing for the correct placement of sound in relation to the paintbrush. A third program, written in C++ and OpenGL, receives position and overall size data from the color tracking program, and draws strokes on a glCanvas based on these parameters.

The stroke's base color is determined directly by the painter in the MaxMSP interface, but the intensity of the color of the stroke, and width of the stroke, is determined by the size of the paintbrush (i.e. the total number of blue pixels found). The painter is in control of the pitch and frequency of the sound via the MaxMSP interface, and the paintbrush is instructed to hold their arms further out to the side when the pitch and frequency of the sound becomes higher. If the paintbrush's arms are further apart, then the color tracking program picks up more blue pixels from the sleeves of the Snuggie, and the size parameter will be increased. A larger size parameter will increase the width of the brush stroke, and the intensity of its color. The painter is also in control of the direction of sound in relation to the paintbrush. The paintbrush attempts to walk towards the source of the sound, and this motion determines the path of the brush stroke. In this configuration, the painter is only given indirect control of what will be painted. The result is determined by the creativity of the painter, the ability of the paintbrush to follow the sound instructions, and the effectiveness of the communication between them.

[*]e-mail:sammat@rpi.edu

2 Previous Work

My own previous method for color tracking with The Human Paintbrush project is handled by a simple and naive algorithm. It scans all pixels in each frame, and for each pixel, it uses a threshold to determine whether or not that pixel is blue. With this information, a blob size and centroid are estimated. Size information is determined based on the number of blue pixels found. A centroid value is then calculated by determining the average of the positions of each blue pixel found.

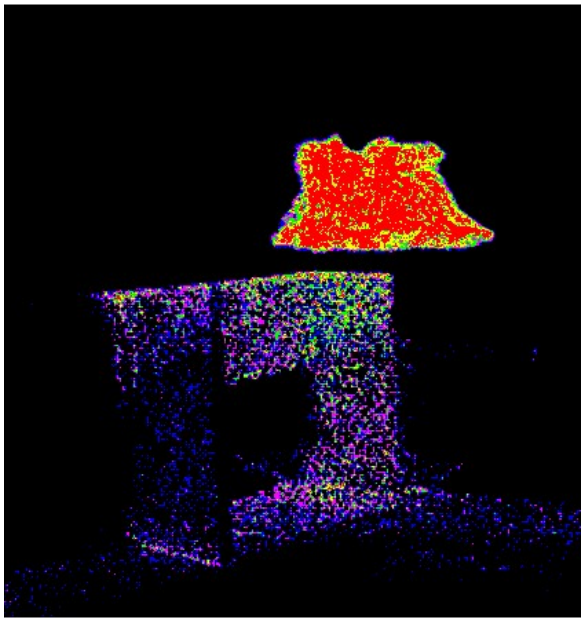


Figure 1: *Histogram debugging information: representing blue pixels found with different thresholds applied. More red colors represent pixels which were blue for higher thresholds*

This simple method proved to be effective for its purposes in a controlled environment, but the threshold for *blue* pixels, which determined how much larger the blue value needed to be compared to the red and green values, needed to be precisely set to ensure proper filtering. It was also very delicate, and small changes in the intensity and color of light in the room greatly affected the resulting number and locations of blue pixels found in the frame. Noise in the filtered frames can also cause problems with the centroid and size calculations. For the duration of this paper I will discuss methods which I have implemented to attempt to correct these issues with my color tracking algorithm for The Human Paintbrush.

3 Normalization

Normalization attempts to remove light source intensity from an image or frame, and leave behind only relational color data. In theory, after streaming video frames are normalized they will not be affected by changes in light source intensity, and the video stream will not vary because of the amount of light in the scene.

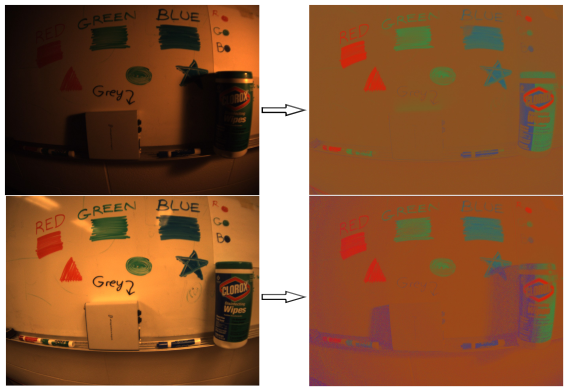


Figure 2: *Canonical input frames adjusted with RG-Normalization. Normalized images differ little, dispite the large difference in light intensity between the two canonical images.*

RG-Normalization, discussed in [Buenaposada et al. 1999], is very simple and works under the assumption that the total intensity for each pixel in the frame is equal to the sum of its red, green, and blue (r,g,b) components. This intensity value is first determined, and then factored out for each of the color channels of each pixel.(1) With this algorithm the resulting vector distance of the (r,g,b) values will always be equal to 1, and hence normalized.

$$(R, G, B) = \left(\frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{B}{R+B+G}\right) \quad (1)$$

As shown in Figure 2, Normalization on unprocessed images is not always perfect. A large flaw with normalizing unprocessed images is that the image or frame might be illuminated by light that is not white. Different colored illumination will result in normalized images which don't accurately account for white and grey colors, even if the intensity of the light source doesn't change. In this case, grey colors have been assigned brown to red values.

4 Color Constancy (White Balance)

Color constancy algorithms are very popular in the computer vision field and in photography in general. Most current digital cameras have a setting for automatic white balance, and many algorithms have been developed in order to solve the problem of non-white illumination conditions. Most of these algorithms work by making an assumption about the overall lighting. They then attempt to estimate the color of the light source, and finally scale the red, green, and blue values for each pixel accordingly.

The *Grey-World* algorithm assumes that the average intensity for each color channel in the frame will be the same. [Lam. 2005] It determines the average intensity of each color channel in the frame (2), and then diagonally scales the channels to ensure that the average intensity for each of them will be the same (3). The green channel is usually chosen to be the referenced channel because humans perceive the color green more intensely than red or blue. Here W is width, and H is the height of the input frame.

$$(R_{avg}, B_{avg}, G_{avg}) = \left(\frac{\sum R}{W*H}, \frac{\sum G}{W*H}, \frac{\sum B}{W*H}\right) \quad (2)$$

$$R = \frac{G_{avg}}{R_{avg}} \text{ and } B = \frac{G_{avg}}{B_{avg}} \quad (3)$$

The *Max-RGB* algorithm works in very much the same way that the Grey-World algorithm does, but it makes a different assumption about the input frame. It ensures that the maximum value of intensity for all the color channels in the input frame will be the same. It determines the maximum intensity of each channel (4), and then diagonally scales the channels for each pixel in the same way as the Grey-World algorithm (5). [Lam. 2005]

$$(R_{max}, B_{max}, G_{max}) = (\text{MAX}(R), \text{MAX}(G), \text{MAX}(B)) \quad (4)$$

$$R = \frac{G_{max}}{R_{max}} \text{ and } B = \frac{G_{max}}{B_{max}} \quad (5)$$

As seen in Figure 3, when tested with input frames that have different colored lighting, the two algorithms both do a good job of white balancing the images, but because the Grey-World algorithm is more consistently colored between different lighting conditions, and because it seems to balance to a more colorless white in my test cases (Max-RGB results in slightly red hue), I chose it as my primary algorithm for attaining color constancy. After further testing the algorithms, such as in Figure 6, it is shown that the best and most stable results, in terms of my own test cases, for making frames light color and light intensity invariant, is to first white balance with the Grey-World algorithm, and then apply RG-Normalization.

After testing the color constancy and normalization algorithms, I noticed that the results I got in terms of the calculation of the number of blue pixels, and the centroid of the blob (the paintbrush wearing the blue Snuggie), were not better, or much different from the results achieved through the naive approach using a properly and precisely set threshold (in fact, some of the results were worse than with the naive approach). The improvement was rather one of consistency. The threshold used for the new approach was constant, and because of the added preprocessing, this threshold was adequate to achieve similar results between input frames with varying lighting conditions.

5 Noise Removal

To further improve upon the color tracking method, I tested two different noise removal algorithms. The motivation for noise removal comes from the fact that no threshold, despite preprocessing with normalization and color constancy, will ever perfectly include every pixel in the desired blob while excluding every pixel not found in the desired blob. The methods currently used to remove noise do so by blob tracking [Park et al.], a method that involves identifying and extracting certain regions of blue pixels within a blob, and

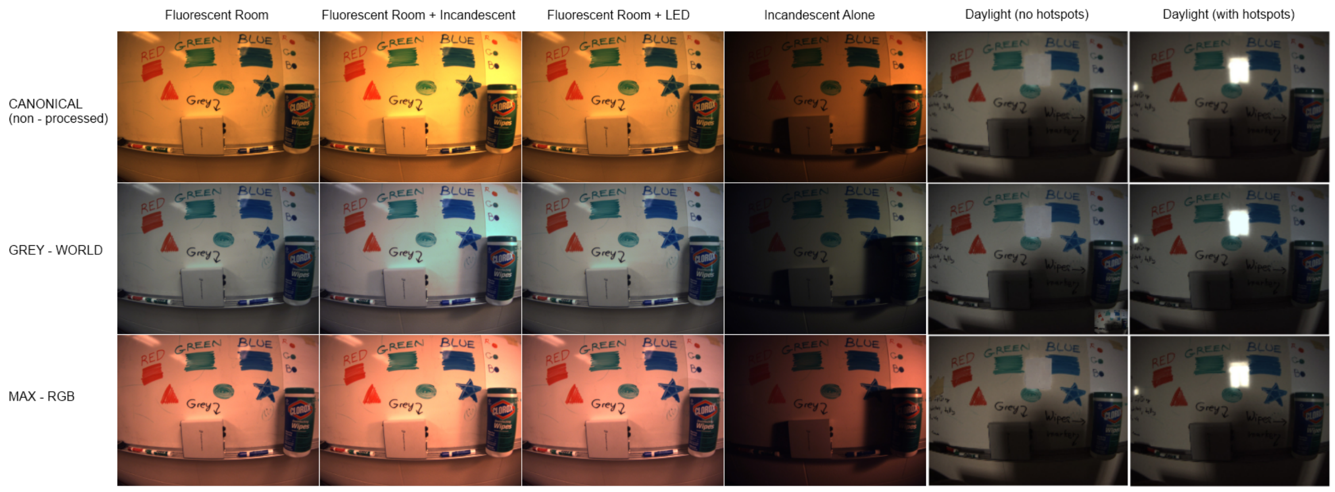


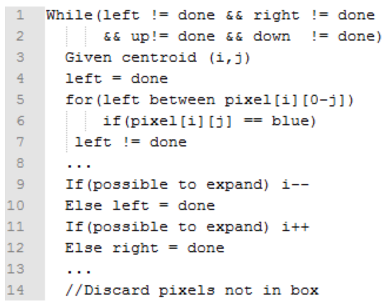
Figure 3: *The Grey-World and Max-RGB color constancy algorithms, applied to input frames with lighting conditions which vary in both color and intensity.*

discarding all other blue pixels as noise. I implemented two different post processing algorithms to accomplish blob tracking; the first method involves fitting a bounding box around a single blob, and the second method, connected components, attempts to extract regions with blue pixels that are next to each other (i.e. share a face).

6 Bounding Box Approach

This algorithm is inspired by the general idea of bounding boxes which, in a two-dimensional field, enclose a region of interest as tightly as possible within a rectangular box. My algorithm makes some assumptions about the incoming data. It assumes that the centroid found with the naive algorithm is within the blob being tracked, and that the point at the centroid is blue. Furthermore, it currently only tracks one blob at a time, and assumes the centroid of the blue pixels will land within the desired blob.

The algorithm functions by starting at the originally calculated centroid pixel. If this pixel is blue, it will attempt to expand each of its four box boundaries (on the top, right, bottom, and left sides) iteratively. Each directional boundary will only be allowed to advance if it has not reached the frame's edge, and if the line formed by the boundary (side of the box, 1 pixel wide) contains at least one pixel which is blue. The final boundaries will not include any blue pixels. The pixels not found within the bounding box are discarded, and a new centroid is calculated based only on the blue pixels within the box.



```
While(left != done && right != done
      && up!= done && down  != done)
  Given centroid (i,j)
  left = done
  for(left between pixel[i][0-j])
      if(pixel[i][j] == blue)
   left != done
  ...
  If(possible to expand) i--
  Else left = done
  If(possible to expand) i++
  Else right = done
  ...
  //Discard pixels not in box
```

Figure 4: *Bounding Box Algorithm (pseudocode)*

Given the appropriate conditions, this algorithm worked very well, and relatively fast (between 20 and 30 frames per second). The largest problem with using this algorithm is that it is not adaptive. Given input that does not fit its specific needs causes it to fail entirely. If there is too much noise in the input, the original centroid may not lie within the blob. Similarly problematic is a blob whose shape is such that its correct centroid lies outside of its colored area. A good example might be a doughnut shaped blob, whose centroid would lie on a non-blue pixel. In these cases, the sides of the box would not advance and the blue pixels in the rest of the frame would be incorrectly discarded.

I considered adding a case which would check for the color of the initial centroid pixel, and if it was not blue, then it would expand until blue is found, and proceed from this point until blue pixels are no longer found. This may have improved the algorithm, but any noise in the initial expansion would have resulted in another totally incorrect bounding box.

7 Connected Components

The connected components algorithm is another blob tracking method, but does not assume as many things about the input as the bounding box algorithm does. It simply needs to be passed a frame for which a threshold has already assigned pixels to be either blue or not blue. The general principle for its operation is to loop through every pixel in the frame, and if that pixel is blue, then a recursive function is called and passed its location and a reference to a data structure holding the pixel locations for the blob. This recursive function starts at the given point, adds itself to the current blob, and then calls itself for the pixels to the right, left, top, and bottom, if those pixels are also blue. The recursion stops when there are no more contiguous blue pixels. The blob with the most pixels is kept, and all other blue pixels are discarded. The centroid and size of the blob can then be more accurately calculated.

As seen in Figure 5, a debugging image shows the largest blobs detected from a particular frame. The blobs are colored on a scale from red, being the smallest of the depicted blobs, to blue, being the largest blob in the frame. The image shows that the Snuggie in the image, colored in blue, was correctly identified as the largest blob, and will therefore be kept. Results for this method of noise removal were far more reliable and accurate than with the bounding

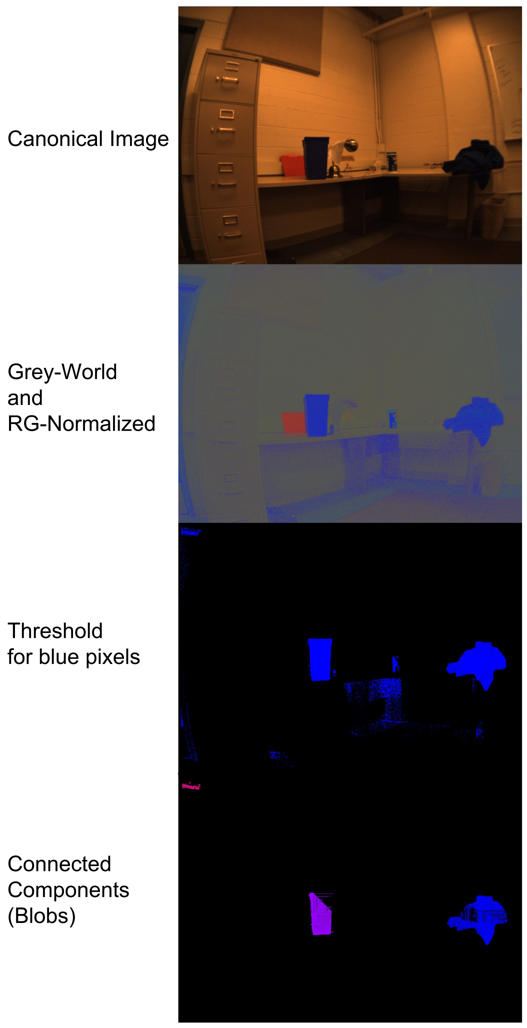

Canonical Image
Grey-World and RG-Normalized
Threshold for blue pixels
Connected Components (Blobs)

Figure 5: *Connected Components Algorithm: Run on a test scene that had already been white balanced (Grey-World) and normalized (RG-Normalization)*

boxes method. It occasionally did not recognize the correct blob as the largest blob, but this algorithm also has the advantage that it can detect and track multiple blobs simultaneously, unlike the bounding boxes algorithm.

8 Problems with Connected Components

The first downside of my connected components algorithm is the fact that my algorithm is not stable because it hasn't been fully debugged. If the blob sizes change too much, then the program has a tendency to crash. I used the color coded output to try to find where the bugs in the code were, but the visible problems, if any, were not obvious ones. The second downside that my algorithm has is that it runs slowly. At first it was too slow to run even in interactive time, and I made a few necessary adjustments to increase its efficiency. The first involved creating a Boolean array to represent every pixel in the frame. Because each pixel can be in at most one blob, once that pixel has been visited, it doesn't ever need to be visited again, and the Boolean array was used to identify which pixels had and hadn't been visited already. This change reduced the running time by a factor of at least N size, where N is the number of pixels in the frame. Another improvement I made to this algorithm's efficiency involved checking only the blue channel of each pixel, rather than all three channels, because my processing before guaranteed that the input frame would be either completely black [0,0,0] or completely blue [0,0,1]. After my improvements were added to the algorithm, it typically ran somewhere in the range of 5-10 frames per second.

9 Conclusion

In this paper I've described two methods for improving upon the color tracking aspect of my already existing project The Human Paintbrush. The first method involves preprocessing the input image in order to make the thresholding process simpler, and the results more robust and constant given different illumination conditions in the source frames. The algorithms I've implemented for these purposes are the RG-Normalization algorithm, to reduce variance due to light intensity, and the Grey-World and Max-RGB algorithms to reduce variance caused by light source color. Through testing I determined the best combination of methods is a normalization of an input which has already been white balanced with the Grey-World algorithm.

The second aspect of improvement involved post-processing noise removal. I implemented two algorithms to accomplish this goal. My bounding box algorithm was effective, but made too many assumptions to be a feasible solution for the project. The connected components algorithm produced better results than the bounding box algorithm, and with less assumptions made about the input, but it was very slow and buggy. I added a few changes to the connected components algorithm to make it run at interactive time.

I completed this project alone. The time spent on the color tracking extensions to The Human Paintbrush was 50+ hours including the time to test and document.

10 Future Work

I would like to spend more time with the connected components algorithm to make sure that it is fully debugged. I would also like to make it more efficient, and add accelerations to bring it into the 10-20 frames per second range. I would like to do more testing with lighting conditions and my color constancy and normalization methods to confirm their ability to create invariant results regardless of illumination conditions. Finally, I would like to create a full working version of The Human Paintbrush with the inclusion of all these changes, and compare it to the previous naive version. A more robust color tracker will allow me to install this project more quickly, and in less controlled environments.

References

EDMUND Y. LAM. 2005. Combining Gray World and Retinex Theory for Automatic White Balance in Digital Photography In *Proceedings of the Ninth International Symposium on Consumer Electronics*, 134–139.

JOS M. BUENAPOSADA, LUIS BAUMELA 1999. Variations of Grey World for face tracking *Universidad Politecnica de Madrid*, 1–12.

JUNG-ME PARK, CARL G. LOONEY, HUI-CHUAN CHEN . Fast Connected Component Labeling Algorithm Using A Divide and Conquer Techniqe *University of Alabama, Tuscaloosa - University of Nevada, Reno*, 1–4.

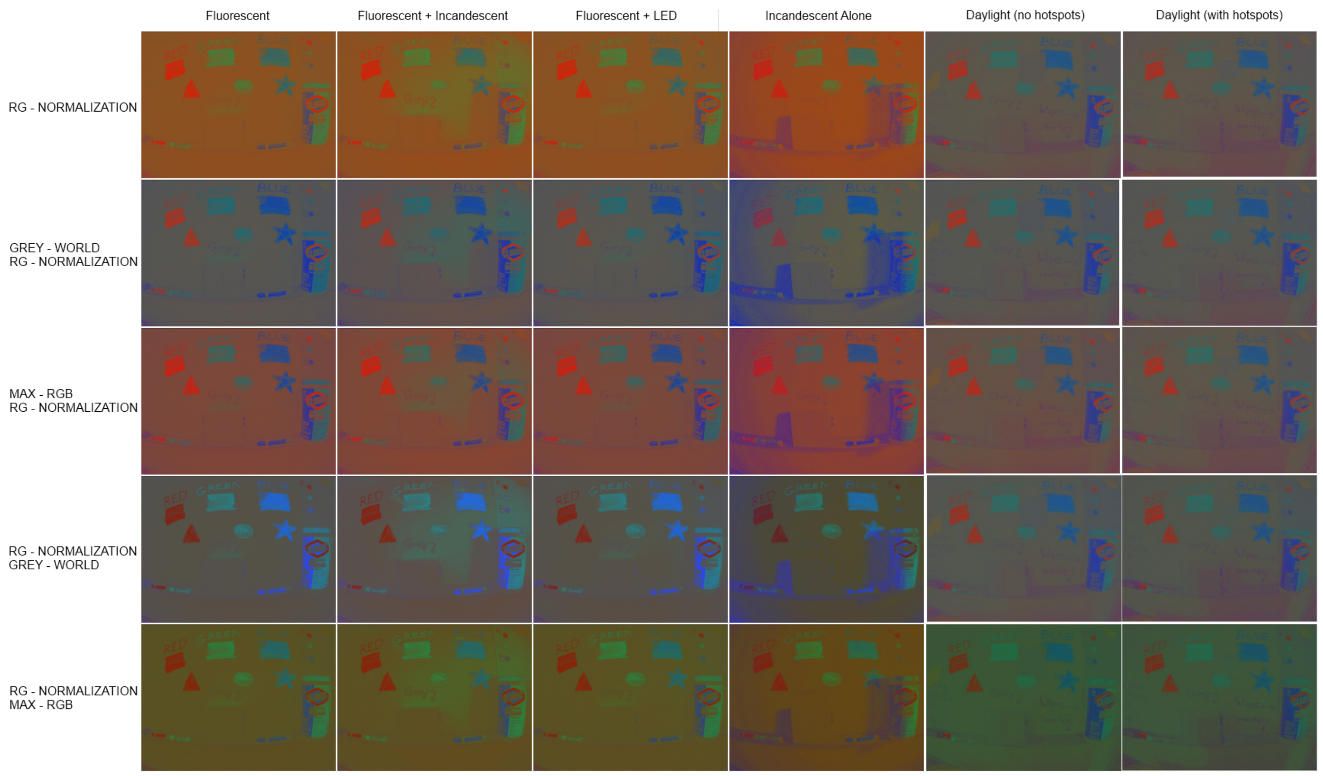


Figure 6: *RG-Normalization applied to canonical, and white balanced test frames with lighting conditions which vary in both color and intensity. Also included are examples of color constancy algorithms applied to normalized frames.*