Analysis of "A Realtime Object Tracking System using a Color Camera"

By: Jacob Scott

UIN: 679325389

Email: jscott35@uic.edu

Github: https://github.com/ece-jacob-scott/object_tracking

Abstract

This paper is a summarization and introspection of an object tracking algorithm described in a paper called “A Realtime Object Tracking System using a Color Camera”, which was presented by three computer scientists: George V. Paul, Glenn J. Beach, and Charles J. These three computer scientists all worked at the time for the CyberNet computing company. According to the “About” section of CyberNet’s website, it prides itself on being able to take the research it creates and bring it to market [2]. Thus, a key feature of the proposed algorithm is its low computational requirement, which allowed it to be adopted by a multitude of platforms in its time. Another goal of the algorithm was to be robust enough to be deployed into production systems. One of the main uses of the algorithm by CyberNet systems was to create a game where the user’s own head was what controlled the game. This system used only the user’s home computer that was equipped with a wired usb camera. A section of this paper will show the results of implementing the algorithm in its simplest form to show it in action using a sample video. The simulation code was written in Python using multiple libraries that exist in the Python ecosystem for working with editing video files and performing matlab style computations. Finally, the performance of the object tracking algorithm using the input from a colored camera described in the article is discussed in depth and compared to the bleeding edge algorithms that exist in this more advanced age of computing that did not exist in 2001.

Introduction

Object tracking software has been the cornerstone of many technologies that are rising in popularity such as gaming, surveillance, and autonomous machines. This need for object tracking solutions has led to a huge effort from the research community and open source communities to come up with algorithms for performing this task efficiently. However, where some of these algorithms fall flat is in their ability to utilize hardware resources efficiently so that you don’t need a powerful machine to perform object tracking. This means that some of the popular algorithms are not the right solution for many products that have limited resources with which to run expensive robust object tracking solutions. This is in contrast to the algorithm described in a paper written in 2001 entitled “A Realtime Object Tracking System using a Color Camera” [1]. In this paper, the authors describe a real time tracking system that utilizes a color

webcam that can run with comparatively low CPU overhead due to the utilization of efficient calculations. The article outlines the use of three distinct calculations that, when used in conjunction with each other to efficiently track an object moving in a video, result in an efficient and robust solution to trivial object tracking.

To give some background to "A Realtime Object Tracking System using a Color Camera", it was created by a company called CyberNet [2]. CyberNet works in the creation of software and, in the early 2000's, they created a gaming platform called "Use Your Head". The design of the platform was simple: the user would use a webcam attached to their computer, and then the player would interact with game by using their head and having the game track the location of the user's head in conjunction with game objects. This type of gaming platform was very ambitious for the time since the average home computer was not as powerful as they are today. Thus, it necessitated a new robust and efficient algorithm for tracking the user using only primitive hardware by today's standards. The resulting algorithm will be mentioned in the following sections, which will describe its usefulness, robustness, and efficiencies.

Recently, the field of research regarding object tracking and detection has made a move from traditional algorithms that require heavy calculations to using machine learning algorithms for massively increased effectiveness. This rise in machine learning technology has been brought about by an increase in feasibility of learning about the topic, scholarly researchers devoting careers to machine learning, and a need for new companies to advance the technology for their products. Many highly involved communities of programmers and researchers have grown exponentially since the discovery of machine learning's abilities due to the ease of learning about the topic, which was made possible by the wealth of knowledge presented by the internet. It is for these reasons that many algorithms have been developed so quickly since 2001, when the paper was written. Even though the main focus of object tracking algorithms has changed gears towards these machine learning style techniques, it is still worth learning about other algorithms that could be used in more trivial systems.

Related Work

"A Realtime Object Tracking System using a Color Camera" was written and researched in 2001. During this time, there were many solutions to object tracking already available.

However, even though there were many solutions, this does not mean that they were useful for every application. Some utilized high cost algorithms to separate out objects from their backgrounds, such as running transform operations or using whole frame point by point comparisons to find object features. One solution, proposed by Ferran Marques and Cristina Molina, shows the ability to track objects using image segmentation, which utilizes operations such as optical flow, edge detection, and texture measuring [3]. Next, their algorithm uses these methods to find the object and then subtract the background, which compares the current frame to a background reference frame that contains no moving objects. After that, the algorithm can segment out all the image parts using various expensive segmentation calculations. Only after all of this preprocessing is done can the object in question be tracked. Obviously, the amount of calculations needed to be executed for this algorithm to track an object would be absurd for applications in a real time system. While an algorithm like this would be useful for non real time object tracking applications, many of the current uses for this technology are in a sector where real time applications are king, which makes this algorithm difficult to use in today's software.

Other algorithms of the time may not require excessive computation time, but are only created for specific applications. The algorithm introduced by Constantine Ponticos was meant to be used for users of handheld phones for object tracking [4]. This algorithm relies on some assumptions such as the assumption that the video will be one of low quality, will have an out-of-focus background while the foreground is clear, and that the subject of tracking will be close to the middle of video and will not move much from frame to frame. These assumptions worked fine for this specific application, but many developers would be hard-pressed to adapt this solution to more general systems. An application of this algorithm would be to track the user of a handheld phone while they were taking a video of someone else or of themselves. This could be utilized with the front-facing camera to track the user, who would most likely be in the middle of the screen. One application for the algorithm would be to apply a filter only to the user being tracked by the algorithm instead of to the whole frame. The algorithm had multiple steps to its object tracking system, starting with obtaining a reference image of the target to utilize in later calculations. Then, the algorithm would search block by block to create a difference criteria. It would next exclude parts of the difference criteria to find the tracked object. While this algorithm

is not as expensive as others of the time, its ability to only track a specific type of object in a specific setting makes the system impractical for all cases.

In 2019, the options for object tracking have made a move towards machine learning and deep learning techniques because that is where a lot of research has focused since 2001 when the paper was presented. This area of research was not as wide and varied during the early turn of the milenia, and thus these types of techniques could only start emerging recently. Before talking about some of the specific techniques utilized in recent times to perform object tracking, it is important to understand the difference between machine learning and deep learning. Machine learning is based on having a specific, well-understood data set that can be reasoned about appropriately, which is then utilized in repetitive calculations to incrementally update an algorithm specified by the developers. While deep learning is a subset of machine learning, which computes over data using algorithms similar to the ones in machine learning but each of them focuses on a different part of the data being supplied. Thus both of these types of algorithms are useful in object tracking and detection, and both have been investigated for their potential uses. The difference between both methods is subtle to people who do not research the field, but each results in a different approach to solving the object tracking problem.

The You Only Look Once (YOLO) algorithm is a deep learning algorithm that was introduced in 2015 [5]. Authors of the algorithm have been able to utilize the approach to track multiple different objects at forty five frames per second on slower hardware, but has been clocked at one hundred and fifty frames per second when optimised appropriately. This metric blows away all of the previously discussed traditional object tracking methods, and it is all possible because the technique does not rely on iteration of expensive calculations that researchers of twenty years ago were forced to use. Not only is this algorithm efficient enough to be used in a real time system, but it is also robust due to the nature of deep learning algorithms. The YOLO algorithm is fast because it reasons about the object detection on a global level, processing the whole image once to make the best predictions. This is in direct competition with other deep learning algorithms and traditional object tracking algorithms that work by solving sliding window problems or by doing block by block comparisons, which require relatively slow computation that grows with image resolution. YOLO moves away from both of these

techniques by seeing the entire image during training, which allows it to encode the image and then it splits the image into separate boxes that may or may not contain the target. Each box is given a probability of whether it contains the target or not, and then the prediction is made. This is an efficient method because it does not rely on comparison or slow, linearly-scaling calculations. Just pass the algorithm an image and it will result in a best guess of where the tracking objects belong. Due to the nature of how deep learning works, these guesses are extremely robust, and similar techniques have even been adapted to work in autonomous driving applications.

As shown, the types of object tracking algorithms have evolved with the times given growing technology and availability of knowledge in the area. During the time that the traditional object tracking algorithms were being developed, tools such as video cameras, powerful computers, and programming language libraries were only available to researchers. With the advent of the cheap video cameras that come attached to every device, the internet, available computer science education, and open source software, these barriers have been severely lowered to the point where the average person can contribute to the developing technologies. Another contributing factor is that the market for systems that require object tracking as a core fundamental function have grown in popularity since 2001. The autonomous vehicles and immersive gaming markets have exploded in the last decade, and these require efficient object tracking algorithms and therefore have pushed the boundaries for these types of algorithms. These factors create a more open and passionate community, which results in more efficient algorithms that are developed more quickly than in past generations. It is these features of the current development environment that result in efficient algorithms at a breakneck pace.

Methodology

Overview

The methodology behind using the real time color object tracking algorithm is simple by design. The algorithm uses a few initial assumptions and a color matching function to calculate three separate center points for the tracked object using three different criteria. When combined, the result is an accurate estimation for where an object is in the current frame. To keep the computation time low, the object tracking algorithm is only run on a subset of pixels in the frame

where the object lies. In the paper, the authors say that the algorithm is robust enough that it only needs to utilize a search area one hundredth of the frame size to track the object [1]. After calculating the current color match values for the frame, the functions for calculating the center of the object can be executed. After taking the average of the center function outputs, the frame has a new center for where the object should be located.

Color Matching

The most integral function for the algorithm is the color matching function, this is the most integral function because all of the other operations in the algorithm are backed by this one calculation. The basic assumption for the color matching algorithm is that the object being tracked is not going to change color very much throughout the video. Given this assumption, we can track an object based solely on its color value compared with the rest of the image. Thus, to get an accurate understanding of where the object is, we run this color matching function in the frame pixel by pixel, and what results is the object itself.

The theory behind the algorithm is simple: think of every color component (red, green, blue) value as being a component of a vector that lies in a three dimensional coordinate plane with the axis being the color component values from zero to two hundred and fifty six (assuming 8-bit color values). This idea is shown in Figure 1. Given the red, green, and blue values of a pixel, one could easily calculate the angle and magnitude of the vector that would correspond to a specific color. The color matching algorithm also requires a reference color, which would be the approximate color of the object. This value is used as a comparison value with the actual color value of the current pixel, and can be obtained through many trivial methods such as sampling the object before the tracking starts and using the sampled value for all future calculations. For the actual algorithm, all of these ideas are brought together in the calculation of a few simple equations. One of these involves finding the magnitude of the projection of the target pixel color and the reference pixel color given as $d_m = \frac{d}{m_r}$, where $d = RR_r + GG_r + BB_r$ and $m_r = R_r^2 + G_r^2 + B_r^2$ (subscript *r* refers to the reference values). The other calculation is to find the angle difference between the target and reference color given as $d_a = \frac{d}{\sqrt{m_r m}}$, where $m = R^2 + G^2 + B^2$ and the other constants are the same as previously mentioned. If these calculated values are within a threshold, then they are said to be good enough for a color match,

and the product of both the angle and magnitude are returned from the function for that target pixel. If the value is not within the threshold, then a value of zero is returned.

These calculations and thresholds essentially make a cone structure inside the three dimensional color space where every point inside that cone is considered a color match. This cone is shown in Figure 1. Here, the idea is visualized and shows that any point that lands in the highlighted area would be considered a matching color. In this example, the point *P* is shown to be a matching color value because it exists within the cone structure. It is important to keep this algorithm computationally simple because it is going to be run every frame for every pixel within the search region and thus takes up most of the computation time. The threshold values are very important to the algorithm because it allows for slight variations in the color value of the tracked object, which increases the robustness of the tracking algorithm. Throughout the video capture, this color matching calculation takes up most of the computational time by a factor of five or higher as compared to the other functions.

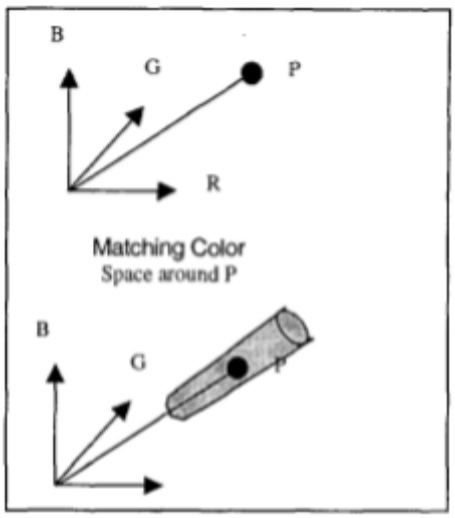


Figure 1: Showing cone of matching points in color space. [1]

Finding Shape and Movement

After finding the object in the current frame, it must be compared to where the object was in the previous frame to decide where the new shape of the object is, and whether the object is moving or not. The calculation for finding the shape of the object being tracked is just a simple pixel by pixel comparison between the color match values from the previous frame and the current frame. At each pixel, if the previous frame had a color match value, then return the current frame's color match value at that point. Otherwise, return a zero to indicate that the point is new for this frame. The output of this calculation is the shape of the object that overlaps between the previous and current frame. Computing the new shape of the object every frame has an advantage over assuming the shape in that it makes the algorithm more robust. It also avoids other shape-generating algorithms that may be more complex or require more computing requirements that are used in other algorithms such as contouring methods.

A similar algorithm to the one used to calculate the shape is used to calculate the motion of the tracked object. Before the actual motion tracking algorithm is run, a motion detection algorithm is used to check if any processing even needs to be done in this step. This is used to reduce the amount of times the motion tracking calculation needs to be run. If motion is detected, then the difference between the two frames is used to find the new target. This is done by computing the difference in the current frame from the previous frame, then isolating all the colored pixels that return from the color match function. The resulting object is going to be used to compute the center in the next phase of the calculation. Figure 2 shows a visual representation of this part of the motion tracking algorithm. The third image in the sequence shows the difference in the two frames, and the fourth image shows what remains after extracting out only the color matched values.

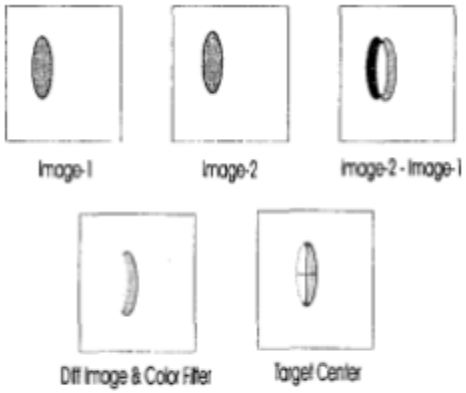


Figure 2: Showing motion tracking algorithm. [1]

The motion tracking algorithm can be summarized using a few simple mathematical functions. First, the algorithm needs to calculate the difference between the two images. This is done by taking the absolute difference between the color match values. If they are higher than a threshold, the difference shows movement (Figure 3 D). The other necessary expression is used to decide whether the current pixel's color match value is above a threshold. This is used to decide if the object being tracked is the actual object the algorithm wants (Figure 3 M). The intersection of these two functions is used to decide whether a point shows movement or not (Figure 3 V).

$$D(i,j,t) = \begin{cases} 1 & if \left| P(i,j,t) - P(i,j,t-1) \right| > \tau_m \\ 0 & otherwise \end{cases}$$

$$M(i,j,t) = \begin{cases} 1 & if (P(i,j,t) > \tau_c) \\ 0 & otherwise \end{cases}$$

$$V(i,j,t) = (D(i,j,t) \cap M(i,j,t)) * P(i,j,t)$$

Figure 3: Motion tracking functions for difference (D), Motion (M), and final result (V).[1]

Calculating the Centers

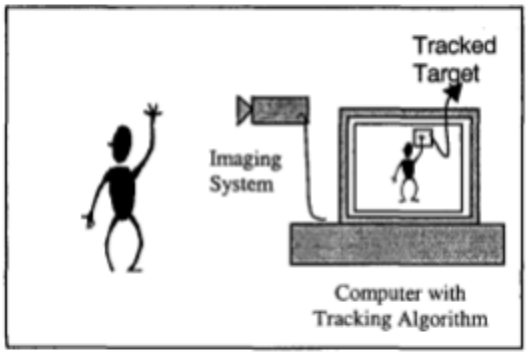
The most important function of this object tracking algorithm is calculating the center points for all the given center points determined for the object. The algorithm needs to calculate three center values every frame: one for each type of calculation done on the object. The centering function is a simple calculation that computes the centroid point of the object given its color matched value multiplied by the row and column where the pixel exists in the frame. This results in a weighted sum expression. This weighted sum expression, when calculated over the entire subset of points that contains the object, results in the center point for that representation of the object. All the centering calculations are shown in Figure 4 for base color value of the object being tracked, the shape of the object, and the motion of the object.

$$Center_{color} = \begin{bmatrix} r_c \\ c_c \end{bmatrix} = \begin{bmatrix} \frac{\sum_{1}^{[r]} P_c(i,j,t) * i}{\sum_{1}^{[r]} P_c(i,j,t)} \\ \frac{\sum_{1}^{[r]} P_c(i,j,t) * j}{\sum_{1}^{[r]} P_c(i,j,t)} \end{bmatrix} \quad Center_{shape} = \begin{bmatrix} r_s \\ c_s \end{bmatrix} = \begin{bmatrix} \frac{\sum_{1}^{[r]} S(i,j,t) * i}{\sum_{1}^{[r]} S(i,j,t)} \\ \frac{\sum_{1}^{[r]} S(i,j,t) * j}{\sum_{1}^{[r]} S(i,j,t)} \end{bmatrix} \quad Center_{motion} = \begin{bmatrix} r_m \\ c_m \end{bmatrix} = \begin{bmatrix} \frac{\sum_{1}^{[r]} V(i,j,t) * i}{\sum_{1}^{[r]} V(i,j,t)} \\ \frac{\sum_{1}^{[r]} V(i,j,t) * j}{\sum_{1}^{[r]} V(i,j,t)} \end{bmatrix}$$

Figure 4: Centering algorithms for each type of object being computed. [1]

Tracking

The holistic algorithm to track an object in a video uses all of the previously mentioned parts. First, the algorithm needs an approximate location for the object needing to be tracked. This can be done in a multitude of ways including setting up a “capture zone” where any object that starts in that zone will be tracked, or by simply inputting initial coordinates for the object if the video is static. Next, the algorithm needs to find and calculate all the shape, color, and motion object color maps for the tracked object. After that, the object’s center can be calculated using the weighted sum of all three individual center values. This, in turn, will give you the new center for the object in the current frame. Lastly, calculate the new tracking location where the object is said to be in the next frame and restart the calculation for the newly captured frame. A flow chart and diagram are provided in Figure 5 to show the outline of this holistic algorithm.



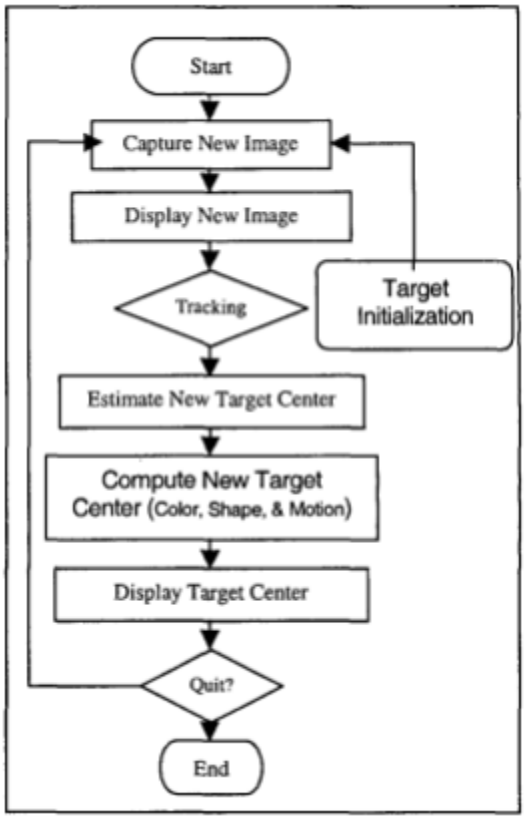


Figure 5: Diagram showing the basic overview of what the algorithm does (top) and flow chart showing the software for the overall algorithm (right).[1]

Simulation Experiment

Overview

To test the algorithm, I decided to implement a simplified version of it using the Python programming language along with a few open source libraries. These libraries I used are called numpy, which was used to assist in matrix manipulations, and opencv, which handled all of the interactions with video files in the project. To decrease the amount of overhead in development, I used a simplified version of the algorithm that includes working only with static video files and updating the tracking algorithm to just use the last calculated center to find the object in the current frame. These considerations that I had affect the overall usefulness of the program, but still accurately demonstrate the effectiveness of the tracking capabilities.

The key to my implementation was representing the image color values as a dictionary object in Python where the x, y coordinates are the keys, and the value at that key is the color match value for the corresponding pixel. This made it easy to compose separate functions for calculating the shape, color, and motion objects for each frame because the color map values are

represented as a well-defined object. Shown in Figure 6 is the function that creates the color map given the subset of the frame where the tracked object lies. The color match function is a function that calculates the color match value based on the reference color and threshold values, which are both global variables for simplicity. This function and dictionary object is the main abstraction used throughout the program and is run on every frame throughout the algorithm's life cycle. Note that the reference color was decided on beforehand given that my test video was static and I knew the color value ahead of time. This value is saved as a global variable named "reference color" which is going to be used throughout the execution of the program and it can be updated at any time by any function.

```
def color_map(frame: np.array, start_x: int = 0, start_y: int = 0) -> Dict[str, float]:
    """
    Creates a color mapping for the passed frame
    """
    m = dict()
    for r, row in enumerate(frame):
        for c, col in enumerate(row):
            key = f"{c + start_x}|{r + start_y}"
            if key in m:
                raise Exception(f"Duplicate key {key}")
            m[key] = color_match(col, REFERENCE_COLOR, DM_THRESH, DA_THRESH)
    return m
```

Figure 6: color_map function which creates the color map for a frame.

Main Loop

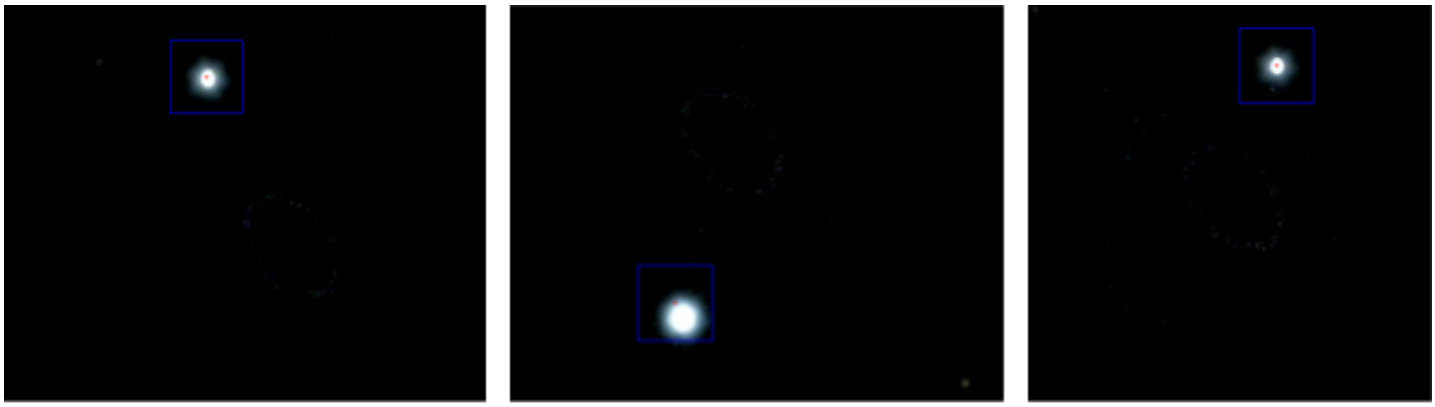
The main loop uses a very simplified version of the algorithm where each center point is calculated separately. This is a more inefficient way of producing center values, but more instructive for demonstration and easier to implement. Note that to make the program easier to read and manipulate for tests, most of the variables used to track the object are global variables defined at the top of the main program file. By convention, these variables are named with all capital letters separated with underscores between separate words. The appendix has the main function in whole, and comments describing what each line of code does in the function. This main loop differs from the original paper's flow chart in multiple ways, but they are all for the sake of simplicity and actually showing off the robustness of the color tracking algorithm. One of the differences is that target initialization and calculating the approximate center is implicit

because each frame uses the center point for the previously calculated frame to know where to start calculating. This implementation detail reduces robustness for the sake of simplicity. This means that the only computation left to do is to find all of the new target color match values and calculate the center of the object using these values. The target is then displayed on the frame, and shows both the calculated center value and the search area that was used to calculate that center value.

Results

Video

To test the algorithm, I found a video online that was made which shows a bright white star moving around a frame with a completely black background. This was the easiest way to test the algorithm because it has the most color contrast with which to identify the tracked object. Figure 7 shows the progression of the algorithm at different points in the progression of the video. In the images, the red dot is showing the calculated center of the object for that frame and the blue box is showing the subset region that was used in calculating the center values.



Beginning Middle End

Figure 7

The results in Figure 7 are shown from the start of the video to the end of the video (left to right). As the video progresses, the star starts at the top left of the frame and does a circle around the frame where, at the end of the first loop, it explodes into a galaxy and the algorithm stops tracking. As shown in the middle image, when the star speeds up, the center point lags behind. This can be helped by switching around some built-in parameters. Shown in Figures 8-10 are the results of running the algorithm over the same video but switching some parameters indicated by their description. The tracking algorithm has lost the object if the target center is located at the

origin point (the top left corner of the image). Note that the best parameters found to work for Figure 7 were fifty percent weight given to the center generated by motion and equal weight between both the color center and shape center, a search size of sixty rows by sixty columns, and a reference color with one hundred and twenty five for all color values.

Performance

To measure the performance of the algorithm, Python has a builtin library of functions called the "cProfile" library that can be used to show how much time is spent in every function during the program's execution. Using this function, the profiler shows that out of the total time spent in the main function during the program's execution— twelve seconds— ten seconds of it is spent in the color matching algorithm. This is because the color matching function needs to be run every frame for every pixel in the subset region. Figure 11 shows the percentage of time spent in the main function. The parameters used for the measurement in Figure 11 were the same used for Figure 7. In total, there were one hundred and eighteen frames in the video, and it took around twelve seconds to complete. This means that a frame can be processed around every tenth of a second using the same parameters described for Figure 7.

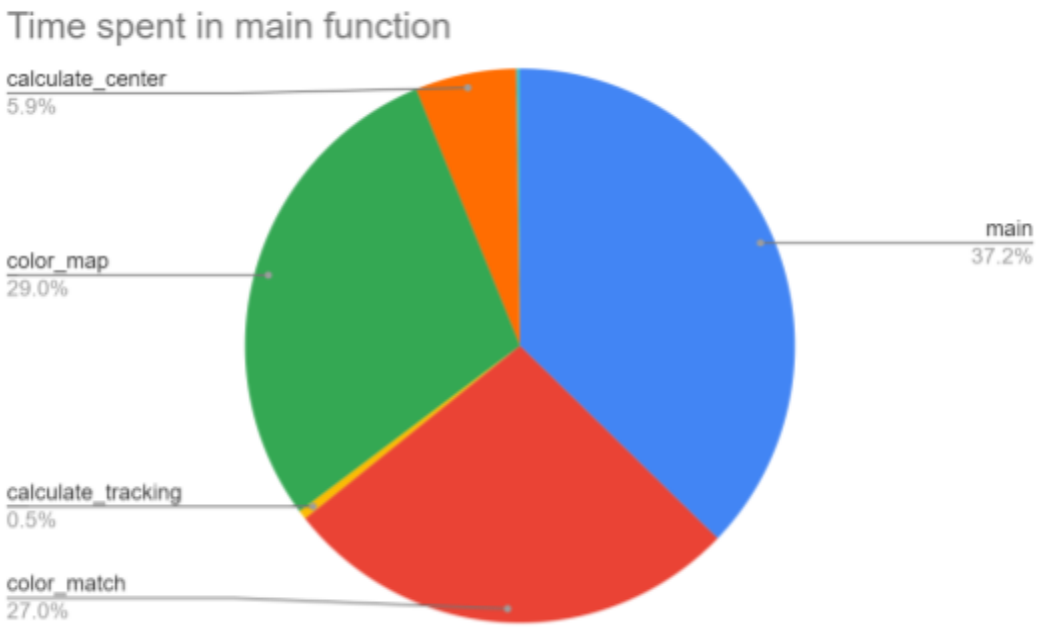


Figure 11: Chart showing amount of time spent during the program life cycle.

Discussion of Results

The results in Figures 8-10 show the outcome of providing different parameters to the object tracking algorithm, as well as computational time for the algorithm using the best parameters for tracking the object. The default parameters used for measurements are fifty percent weight given to the center generated by motion, and equal weight between both the color center and shape center, a search size of sixty rows by sixty columns, and a reference color with one hundred and twenty five for all color values. Figure 7 uses the default parameters, and shows frames at different time intervals to show that with the default parameters, the object is tracked for the duration of the object's movement. The default parameters are used because they provide the best outcome for tracking the star as it moves in the picture. They provide the best outcome because the reference color was gathered from sampling the object and extracting the average color before the video is run. The dimensions of the search area were decided on by experimenting with different values and running the experiment multiple times. The smallest dimension that tracked the object best was sixty rows by sixty columns. The dimensions are square because the fact that the start rotates in a full circle around the frame means that it does not move in any direction more than another, so a square provides the best search shape. Center weights were decided heuristically and experimentally by trying multiple different values to see what gave the best result. As the object moved further between frames, the targeting algorithm would fail to find the new object and proceed to lose the object as a result. This is why the center point found by the motion calculation has a higher weight to compensate for the object gaining velocity as it moves through the video.

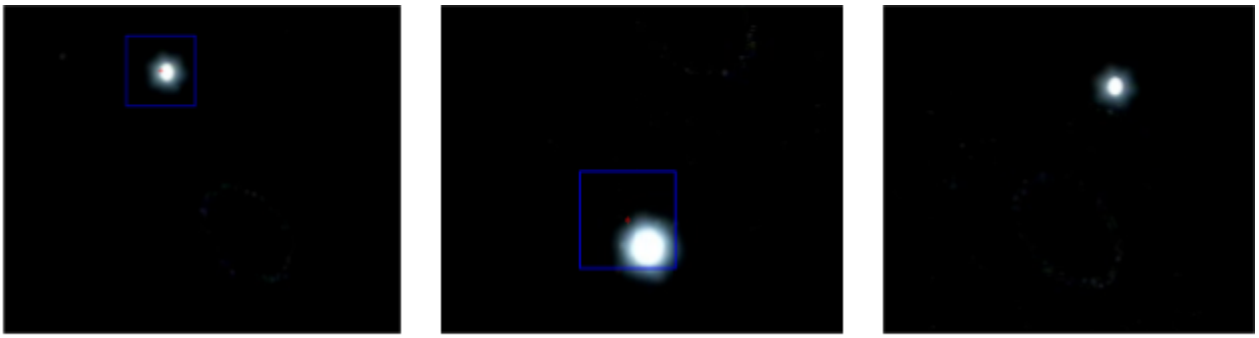
Results from Figures 8-10 show the difference in the algorithm when the default parameters are changed one at a time. In Figure 8, it tests the values of the center weights and shows that if the centers calculated by all three functions are weighted equally, the object will be lost once the object starts moving more between frames towards the end of the star's path. This is why for fast-moving objects, the center created by the motion calculation should be weighted higher to compensate for a fast-moving object. Also, note that if the object is moving fast enough between the frames, the shape calculation will no longer be relevant because the overlap between the object in the current frame and the object in the previous frame does not exist, which results

in no shape able to be calculated. This would mean that for objects with an increased velocity, the center created by the weight should not be taken into account because it cannot be accurate due to a lack of available overlapping points.

Center Weights:
33% Motion
33% Color
33% Shape

Search Size:
60 rows
60 columns

Reference Color:
(125, 125, 125)



Beginning | Middle | End (lost the object)

Figure 8

Center Weights:
50% Motion
25% Color
25% Shape

Search Size:
40 rows
40 columns

Reference Color:
(125, 125, 125)

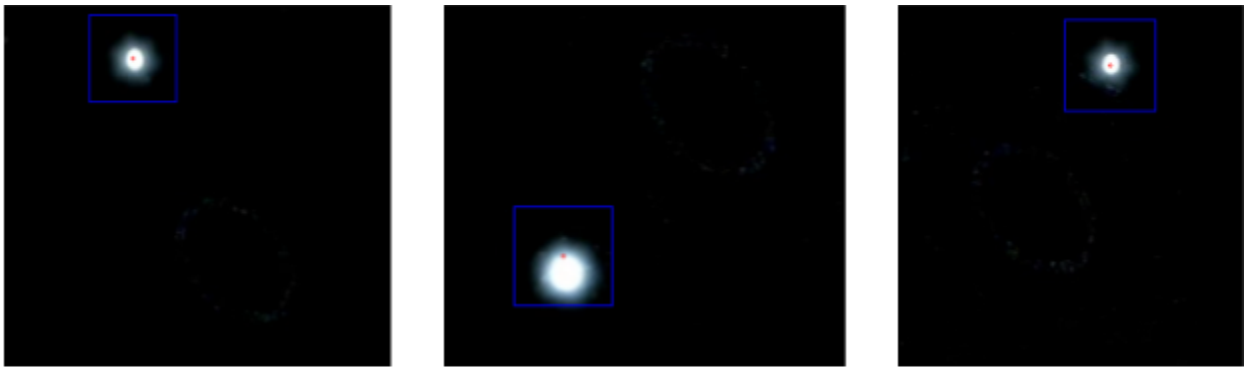
Beginning | Middle (lost object) | End (lost object)

Figure 9

Center Weights:
50% Motion
25% Color
25% Shape

Search Size:
60 rows
60 columns

Reference Color:
(40, 40, 40)



Beginning | Middle | End

Figure 10

Figure 9 shows the results of a decrease in search size. The decrease makes the algorithm stop tracking the object as the object moves further between the frames. If the object moves

enough pixels between frames, then it will fall outside of the search area range, and this results in the algorithm losing the object entirely. One way to mitigate this would be to raise the weight of the center value created by the motion calculation. I could not, however, get this to work in testing with higher weight values, but in theory it should work out that way. My simulation did not work with this higher weight value because of the way I calculated the starting point of the object. Using the previously calculated center value is not as robust as finding the tracking target every frame, and resulted in the object being lost if it moved far enough between frames. One advantage to dropping the search is that the execution time per frame drops linearly with search size. This is obvious because the color mapping process runs for each pixel in the search area for every frame, and if there are less pixels to color map, then the execution time of that function decreases. This means that for future applications, trying to minimize search area size should be the main priority as it is so closely related to execution time.

Figure 10 shows the result of lowering the color value, which resulted in little change unless brought to the extremes either by making the color components close to zero or close to two hundred and fifty five. From testing, it seems that as long as all the individual color values are the same, because the color of the tracked object is white, then the range of color values where the object is still able to be tracked is between forty and two hundred for each component. The reference color value would have a higher correlation to tracking performance for videos with a higher variety of color. The lack of color in the tested video is the reason that changing the reference color value had very little effect on the final result. However, the algorithm still works the same given a mostly black and white video because the extraction of every pixel is still given as a *24*-bit RGB color value.

Finally, the computational performance of the algorithm was the main consideration for using the algorithm because the authors of the paper originally designed the algorithm to be fast. Looking at the results from running the object tracking algorithm on the star video using the default parameters previously mentioned, each frame is processed in a tenth of a second. This value is gathered from counting the amount of frames in the star trail— one hundred and eighteen frames— divided by the amount of time spent processing, which was around twelve seconds. Also, most of the computing time was spent in calculating the color match value talked

about in the methodology section. See Figure 11 for the whole breakdown of where the algorithm spent most of its time during execution. The implementation provided in the appendix for the color match function performs the exact calculation described in the article. Some considerations should be made for the language used to implement the tests. Python is known for not being the most optimal language for high performance algorithm execution, and was chosen in this paper for ease of development. If the algorithm was implemented using a compiled language such as C, which allows almost infinite levels of computational optimisation, then the execution time per frame is likely to drop dramatically and thus should be utilized in any real application of the described object tracking algorithm.

Future Work

Object tracking has been an integral part of many applications, and thus has been optimised and better figured out over the years. In modern times, object tracking has been integral to many people's lives with the advent of autonomous vehicles. Companies such as Tesla and comma.ai have been bringing the science fiction of autonomous driving to people's garages off the back of efficient object tracking and recognition algorithms. Machine learning advances held the breakthroughs that autonomous driving technology needed, and it was machine learning that held the biggest yields when it came to object tracking algorithms. The advances in the machine learning field over the past few decades have only been possible with a huge surge of support from the open source community. The open source community is going to drive the future work in the field of object tracking.

Open source communities have led the software field forever, and have been credited with many advances in the world. Most recently with the advent of Python becoming one of the most popular programming languages [6], with its multiple well-written data libraries, it has empowered developers to create more than ever. This community is where the future of algorithms in object tracking will exist and flourish. It has already given the world many advancements in the field of machine learning, and Python is the primary programming language for companies that are spearheading the advancements in the new thriving field of autonomous cars. A newcomer to the autonomous vehicle market is a company called comma.ai, which makes products that can be equipped to a user's vehicle and will allow it to have autonomous

driving capabilities. This company has over seventy open code repositories on GitHub, which is the most popular open source website [7]. Comma.ai, being a self-driving car company, runs off of the back of computer vision programs that utilize machine learning algorithms that enable object tracking. Comma.ai also has one of the only products in the diy self-driving market, and is pushing innovation with the help of open source communities.

In the past, researchers at universities have been the source of massive advancements where the common representative in an open source community just do not have the expertise. Many of the most advanced machine learning algorithms have come out of universities being created by individuals who have spent entire careers dedicated to the field of advanced algorithms. This could be another place where object tracking algorithms could flourish in the future, and will hopefully spearhead the search for efficient performing non-machine learning algorithms for lower-powered machines where machine learning is not an option. Wherever the advancements come from in this interesting and exciting field of computer vision, the only certainty given the past is that the most popular algorithm will be one that is the most efficient in its use of computing power.

Conclusion

In conclusion, based on basic tests using a simplified version of the algorithm, the algorithm discussed in "A Realtime Object Tracking System using a Color Camera" can be considered effective. The original use for the described technique was to be used for gaming purposes by allowing for a user to control parts of the game by tracking the player's body parts and relaying that information to the game system. Figure 12 (left) shows this ability in action from an example provided by the original authors, which shows that the algorithm can be used to track a player's hand movements. This example provides insights into the viability of using the tracking algorithm for practical

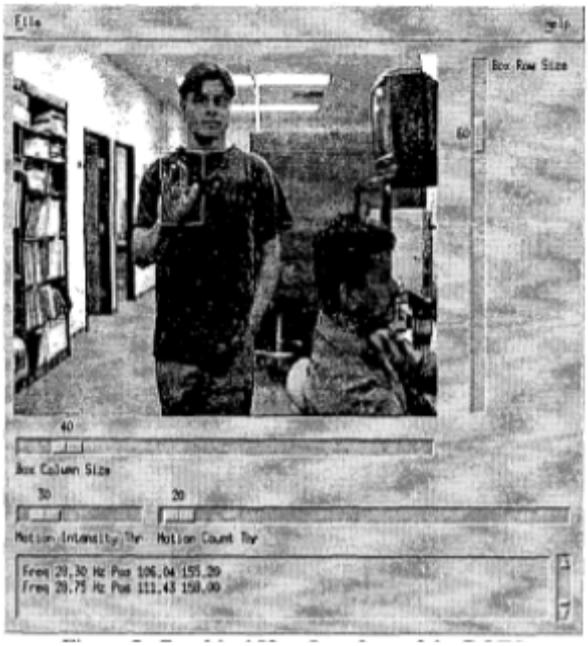


Figure 12: A frame showing the final product described by the article. [1]

purposes. In my experiments, however, it seems that if the wrong parameters are decided on during the program's execution, the algorithm can either slow down, causing lag in the gaming experience, or not be able to keep up with objects that move quickly, this decreases the amount of applications available. In the modern world of technology computation is very cheap compared to what it was in 2001. This means that there is more leniency in using more expensive algorithms for tasks such as object tracking in applications. This attribute of modern computers means that the most impressive characteristic of the paper's algorithm are not as useful anymore, and thus should not impact a developer's decision of whether to utilise the algorithm as much. Developers of today should be more inclined to use more modern algorithms for their current applications given that better algorithms exist in the modern era.

However, even if the developers of this modern era should choose a newer, more performant algorithm for their applications, the system described in the paper still delivers in terms of performance and robustness when utilized correctly. In 2001, when the paper was released, the authors said that they were able to develop an application that could work at thirty frames per second. While this is an almost comically low frame rate, this is also considering hardware that was available in 2001. If developed using a lower level language such as C, with parameters chosen properly for tracking, and if the computer was being backed by a modern GPU chip, the algorithm could be adopted to perform its original purpose in this modern era of gaming. If these considerations are taken into account the algorithm described by the authors could be apart of a modern application where it's robustness and low computation cost are valuable assets.

Finally, given all the information about the object tracking algorithm using a color camera presented by Cybernet researchers George V. Paul, Glenn J. Beach, and Charles J. Cohen, it is safe to assume that even though the algorithm was good for it's time, it has no place in applications that require object tracking today. Software written recently that requires object tracking as a primary function include autonomous vehicles and gaming, both of which have better alternatives than the presented algorithm. Autonomous vehicles, for instance, need to be able to detect multiple objects at once both quickly and accurately, while also being able to recognize what an image is. All of those requirements are not available with just a simple, single

object tracking algorithm. Modern gaming is no different. Most modern games that use human interaction for controls have moved onto other technologies such as virtual reality, or use specifically-designed remotes that can more easily be mapped to games, rather than requiring the developer to think about all of the variables in utilizing just a camera. Thus, while this algorithm was useful for its time, it cannot compete with the latest algorithms developed in the twenty first century in terms of resourcefulness for modern applications and, in some cases, also computational efficiency.

Appendix

Works Cited

[1]Paul, G.v., et al. "A Realtime Object Tracking System Using a Color Camera." *Proceedings 30th Applied Imagery Pattern Recognition Workshop (AIPR 2001). Analysis and Understanding of Time Varying Imagery*, 2001, doi:10.1109/aipr.2001.991216.

[2] "History and Vision." *Cybernet*, 2019, www.cybernet.com/about-cybernet.

[3] Marques, Ferran, and Cristina Molina. *Image Segmentation and Object Tracking Method and Corresponding System*. 10 Oct. 2000.

[4] Ponticos, Constantine. *Apparatus for Tracking Objects in Video Sequences and Methods Therefor*. 7 Mar. 2000.

[5] Redmon, Joseph, et al. "You Only Look Once: Unified, Real-Time Object Detection." *Arxiv*, University of Washington, 9 May 2016, arxiv.org/pdf/1506.02640.pdf.

[6] "Stack Overflow Developer Survey 2019." *Stack Overflow*, 2019, insights.stackoverflow.com/survey/2019#technology.

[7] "Comma.ai." *GitHub*, 2019, github.com/commaai.

Code

```
# MAIN.PY
# Attempt at recreating basic algorithm described in paper
# - https://tinyurl.com/tdtrwkw (need IEEE access)

import numpy as np
import cv2 as cv
from math import sqrt
from os import path, getcwd
from pprint import pprint
from typing import List, Union, Tuple, Dict
from utils import color_match, calculate_shape, calculate_center, calculate_tracking

VIDEO_FILE = "videos/grb_2.avi"
SCRIPT_DIR = getcwd()
# Got the reference color from finding the comet starting position
# [80:125, 290:330] and taking the average color based off a threshold of any
# average color value above 50.
REFERENCE_COLOR = (125, 125, 125)
# Found these threshold values by calculating the color match formula over the
# starting range and taking the average of what I say from the da & dm values
DA_THRESH = (200.0, 400.0)
DM_THRESH = (200.0, 600.0)
# Initial center point (x, y)
INIT_CENTER = (309, 96)
# Height and Width of interest area
```

```
WIDTH = 60
HEIGHT = 60
# Center weights
M_W, C_W, S_W = 0.50, 0.25, 0.25


def color_map(frame: np.array, start_x: int = 0, start_y: int = 0) -> Dict[str, float]:
    """
    Creates a color mapping for the passed frame
    """
    m = dict()
    for r, row in enumerate(frame):
        for c, col in enumerate(row):
            key = f"{c + start_x}|{r + start_y}"
            if key in m:
                raise Exception(f"Duplicate key {key}")
            m[key] = color_match(col, REFERENCE_COLOR, DM_THRESH, DA_THRESH)
    return m


def add_target(frame: np.array, center: Tuple[int]):
    [I_X, I_Y] = INIT_CENTER
    H_W, H_H = WIDTH // 2, HEIGHT // 2
    cv.circle(frame, center, 1, (0, 0, 255), 1)
    cv.rectangle(frame, (I_X-H_W, I_Y-H_H),
                 (I_X+H_W, I_Y+H_H), (255, 0, 0), 1)
    return frame


def main():
    # Create a video capture object
    cap = cv.VideoCapture(path.join(SCRIPT_DIR, VIDEO_FILE))
    # Initialize previous frame map variable
    prev_frame = None
    global INIT_CENTER
    # Get the height and weight for the search area
    H_W, H_H = WIDTH // 2, HEIGHT // 2
    while cap.isOpened():
        # Get the approximate center point for the object
        [I_X, I_Y] = INIT_CENTER
        # Read the current frame
        ret, frame = cap.read()
        # If the reading the frame failed break out of the loop
        if not ret:
            print("Failed to read frame")
```

```
            break
        # If the user pressed 'q' break out of the loop
        if cv.waitKey(1) == ord("q"):
            print("Stopped")
            break
        # Strip out the object search area using the initial
        # points and the height and width values
        sub_frame = frame[I_Y-H_H:I_Y+H_H, I_X-H_W:I_X+H_W]
        # If there was no previous frame then create one
        # and move onto the next frame
        if prev_frame is None:
            prev_frame = color_map(sub_frame, I_X-H_W, I_Y-H_H)
            continue
        # If there was a previous frame calculate centers
        curr_frame_map = color_map(sub_frame, I_X-H_W, I_Y-H_H)
        c_center = calculate_center(curr_frame_map)
        shape = calculate_shape(curr_frame_map, prev_frame)
        s_center = calculate_center(shape)
        motion = calculate_tracking(curr_frame_map, prev_frame)
        m_center = calculate_center(motion)
        # Find center of the object using the weighted sum of
        # all the object's calculated centers.
        n_r = int((c_center[0] * C_W) +
                  (m_center[0] * M_W) + (s_center[0] * S_W))
        n_c = int((c_center[1] * C_W) +
                  (m_center[1] * M_W) + (s_center[1] * S_W))
        # Set the current frame to the previous frame for the next loop
        prev_frame = curr_frame_map
        # Update the initial center value
        INIT_CENTER = (n_r, n_c)
        # Add the target around the tracked object
        frame = add_target(frame, (n_r, n_c))
        # Show the frame with the target
        cv.imshow("Video", frame)
    cap.release()
    cv.destroyAllWindows()


if __name__ == "__main__":
    main()

# UTILS.PY
import numpy as np
import cv2 as cv
from math import sqrt
```

```
from typing import List, Union, Tuple, Dict


def color_match(pixel: List[int], reference: List[int],
                dm_thresh: Tuple[int] = (0, 1000), da_thresh: Tuple[int] = (0, 1000)) -> float:
    [R, G, B] = pixel
    [Rr, Gr, Br] = reference
    d = (R*Rr) + (G*Gr) + (B*Br)
    mr = Rr ^ 2 + Gr ^ 2 + Br ^ 2
    m = R ^ 2 + G ^ 2 + B ^ 2
    # So the algorithm doesn't divide by 0
    if m < 0.1:
        m = 0.1
    dm = d / mr
    da = d / sqrt(mr * m)
    [dm_l, dm_h] = dm_thresh
    [da_l, da_h] = da_thresh
    if dm_l < dm < dm_h and da_l < da < da_h:
        return da * dm
    return 0.0


def calculate_shape(c_color_map: Dict[str, float], p_color_map: Dict[str, float]) -> Dict[str,
float]:
    """
    Calculate shape of the figure using the current frame and the previous
    frames color value

    Algorithm:
    ----------
    if color existed in previous frame then keep the color value
    else set color value to 0
    """
    s = dict()
    for key, value in c_color_map.items():
        if key in p_color_map and p_color_map[key] > 0.0:
            s[key] = value
            continue
        s[key] = 0.0
    return s


def calculate_center(m: Dict[str, float]) -> List[int]:
    """
    Given a dictionary map of an image calculate the center point
```

```
    The dictionary is given as the [x|y]: color_match(x, y)
    """
    x_coor = 0.0
    y_coor = 0.0
    c_total = 0.0
    for key, value in m.items():
        [x, y] = list(map(int, key.split("|")))
        x_coor += (value * x)
        y_coor += (value * y)
        c_total += value
    try:
        return [int(x_coor // c_total), int(y_coor // c_total)]
    except:
        return [0, 0]


def calculate_tracking(c_color_map: Dict[str, float], p_color_map: Dict[str, float], d_thresh:
int = 0, c_thresh: int = 0) -> Dict[str, float]:
    """
    Calculates the color map for tracking the figure.

    Algorithm:
    If the differnce in color is above a threshold and the current color value is
    above a threshold then keep the color.
    """
    v = dict()
    for key, value in c_color_map.items():
        if key in p_color_map:
            difference = abs(c_color_map[key] - p_color_map[key])
            if difference > d_thresh and c_color_map[key] > c_thresh:
                v[key] = c_color_map[key]
                continue
        v[key] = 0
    return v
```