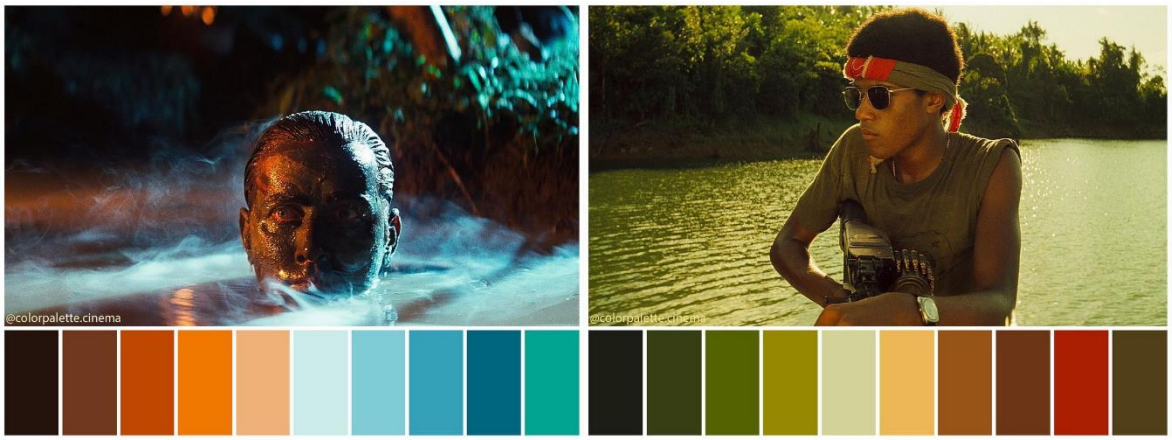
Joseph Michael Coffey IV – jmichaelc4@gmail.com – jmcoffey@colostate.edu

Hayden Corbin - haydenfcorbin@gmail.com - hayden.corbin@colostate.edu

Reilly Bergeron - reillybergeron@gmail.com - reillyjb@colostate.edu

Introduction

The original inspiration for our raspberry pi project came from an interest in the color theory behind compositions in cinematography and photography. There are various instagram pages dedicated to showcasing the use of color in film, with @colorpallete.cinema being one of the more popular ones, having almost 2 million followers as of the creation of this document. Below is an example of the kind of images this account posts, showcasing the color palette of two different scenes from Francis Ford Coppola's 1979 film, Apocalypse Now.



Being a fan of these breakdowns of color usage in film composition, the initial idea for the project was using a raspberry pi in combination with a camera to create this effect on a video feed in real time. The corresponding colors at the bottom of the screen would update to reflect what is being shown to the camera. This could be used as a helpful tool by videographers and photographers to help them get the color palette they want while shooting. However, what if the colors present in the scene aren't satisfactory? The greens need to be more vibrant, the blacks need to be darker, the whites need a warmer tint to them. That was what inspired us to not only create the before mentioned color analysis on a live feed, but also allow the user to then select those colors and edit them in real time as well. This would be a powerful tool for a photographer, as normally these changes couldn't be made till after a shoot was already completed, and the photos / video were loaded into an editing software. We would later come to call this program, Real Time Canvas.

Problem Characterization

We have an idea for our project, but how should we go about balancing the hardware limitations of the raspberry pi with the requirements of real-time video feed and pixel manipulation? The identified components were obviously the raspberry pi, but also a camera as our sensor, which we would need to capture the frames to analyze, as well as a desktop to display our python program. Using these devices, we record the ten most common colors from an original real-time video and make changes to them to customize the coloring of said video. The camera was used to take a live video, the frames would then be analyzed on the raspberry pi, and two copies of the video feed would be displayed on a desktop. One of

these feeds shows the original live video, while the other shows the same video with the user selected color changes. From the first frame of the video, the ten most prevalent colors were recorded and displayed on ten buttons at the bottom of the screen, as well as input fields representing the RGB values of said colors. To edit the colors shown on the second feed, the input fields containing the default color values would be adjusted to represent the desired color. Once the button with the new input field is clicked, the color changes are applied and displayed on the desktop. But it was also possible that the setting in which the camera recorded its live video could have changed dramatically, so we needed some way to address this to make it convenient for the user. This is done through a refresh button, which refreshes the colors to match the feed again. This is all a lot of information for the user to digest, so the UI had to be well designed. The UI used was chosen as it conveys all the information to the user in an easily understandable yet visually appealing way, while also being visually similar to the original instagram posts that inspired our project.

Proposed Solution and Implementation Strategy

We divided the project up into several different sections: the creation of the GUI, the functionality of the program, and the configuration of the Raspberry Pi. The first step was to create a GUI for the necessary components, which included the videos, labels, buttons, and input fields. After an initial draft of the GUI was created, the implementation of the functionality could begin. Before we started development, we sat down to organize our thoughts and plan initial implementations. For our GUI we chose the Tkinter library as “Tkinter provides a fast and easy way to create GUI applications” (TutorialsPoint). Tkinter’s ease of use with element packing and grid design, as well as a flexible library of widgets like buttons and sliders, allowed us to confidently adopt this GUI library. To be able to create an effective user interface, we had to learn about the various methods, which we did through the website TutorialsPoint by experimenting with the provided methods and widgets. Soon, we formed our plan for the interface: a header with a refresh button, two side-by-side frames containing the original video and the changed video, and a frame on the bottom that contained the color palette. At first, we used the method “pack()” to put everything together in our window. We would then work on the functionality and adjust the interface as we saw fit.

We started with two video inputs running side by side in the GUI at once, one “original” and the other “changed”. We discovered this was an inefficient implementation, so instead we determined we could run one video input and just copy the current frame, creating an “original” and “changed” frame without two video inputs. We created a separate process that allows the video input to run as a producer and the main GUI to run as the consumer of those frames, adjusting their size as needed. While testing the color changing functionality, we encountered a bug in the implementation of our programmatic color palette. Due to the way that button behavior is declared in tkinter, our initial implementation caused the behavior of the existing buttons to be overwritten by the next button being declared, resulting in only the last color being recognized. After some iteration, we created a class that defined the behavior of one button-slider-label group to store the slider information, button behavior, and color information for the label, for each individual group. This approach enabled the modular and programmatic behavior that we strived for in the beginning, while abstracting away from the specific implementation in DisplayGUI. During this process, we also made the decision to switch from one text field to 3 RGB sliders to allow the user to change the color value of the button. This decision was made because the user would have been required to memorize all of the names of the colors supported by tkinter with the text field implementation, which is not realistic.

Once we had implemented much of the functionality, we had to make final decisions for the UI. At first, we failed to consider that not every computer that ran the program would have the same screen size, making our interface inconsistent across all desktops. We also failed to consider that the user would decide to adjust the window size. Because of this and our GUI implementation, the widgets would not

adjust accordingly if the window did change. To account for these shortcomings, we decided to have the dimensions of the GUI be slightly smaller than the full screen. We also created formulas for the sizes and positions of the widgets to follow suit. From the website TutorialsPoint, we learned both how to get the screen size and how to disable window adjustment. We disabled window adjustment to ensure consistency in the size and convenience of using the GUI across all desktops, including the one used by the Raspberry Pi.

Even after we solved the issue of the window size, we still had to adjust our frames, as the videos in the side-by-side frames did not use the full frame despite our specifications. We decided to adjust our packing method to use the "grid()" method instead. According to TutorialsPoint, this method organizes each widget packed by this method on a row and column basis. We also learned from the website plus2net about how to change the sizes of our rows and columns – and frames by extension – to best fit our UI. The adjustments we made also cleared up some inconsistency issues with our color frame palette as well as eliminated the need for two side-by-side frames, so we were able to clean up our interface and reduce the amount of necessary memory for the program. We still used the "pack()" method on our non-frame widgets because we packed those into the frames.

We struggled to find code that could detect multiple colors in a video at a time, much less the ten most common ones, before finding an article written by M. Rake Linggar A. Linggar presented three methods for collecting the most common colors. The article covered how to detect colors in an image, but since the captured frames were technically images, we were confident that our adaptation of one of Linggar's solutions would work. Out of the three methods he presents, the third method, and the one we chose to implement, was finding the ten most common colors in a video by using "the infamous K-Means Clustering to cluster groups of colors together" (Linggar). Kevin Arvai, who wrote an article covering K-Means Clustering, described it as "an unsupervised machine learning technique used to identify clusters of data objects in a dataset." To summarize, data objects are organized into certain groups as specified by implementation, which in our case, was colors. There are a few different techniques surrounding K-Means Clustering, but the basic idea for our implementation is that similar colors were grouped together based off of a range that we set. But in order to use this algorithm, we had to download the sklearn module. We also downloaded the collections module to store the cluster in an array. What we returned was an array of ten elements, and the elements were arrays with three numbers – one representing each of the three required values to form an RGB value. But those values were reversed, so we had to treat it like a BGR value. Regardless, our algorithm worked. There was one last problem in our collection of colors; collecting colors for every single frame would severely slow down the video thread, which is why a refresh button was implemented. We decided that only the first detected frame would suffice when it came to color collection. If the refresh button was clicked, the colors would be recollected from the latest frame, and all of the changes made to the live video in the right of the window would be reverted. This solved the issue of a live video with a slow frame rate. Once that was finished, all that was left was to implement the rest of the functionality.

While we aspired to have a clean and functional interface for our program, there is a lot that goes on under the hood to enable the Real Time Canvas to function properly. As the collection of video is the driving force behind the rest of the logic of our program, we decided to go with the python library OpenCV, as this is a common python library for processing external input in general. Compared to other video processing libraries, OpenCV provided the speed of video capturing that we required and could be used alongside other image libraries. We wanted to make sure that processing and interprocess communication was fast enough to maintain an acceptable real-time feel, so multiprocessing was necessary. To enable this process, we created the RealtimeVideo module with the run() function that handles all of the OpenCV overhead and outputs each captured frame to a multiprocessing Queue. The use of queues within this program is critical, as the included Python libraries provide thread-safe queue implementations that are convenient to use and enable fast and effective communication between the

various modules of our program. Besides the frame queue, RealtimeVideo also requires an RGB queue and event queue to pass information about the current color clusters to DisplayGUI and to receive events from DisplayGUI that need to be handled within the RealtimeVideo module.

Touching briefly on the event queue implementation, this is used to support the functionality of the “Refresh” button, which collects new common color clusters and resets the changed color panel within the GUI. To enable DisplayGUI to talk to RealtimeVideo, we created an event queue that takes in an event in the form of an event name and parameters, then executes that event based on predefined behavior. We needed to create a helper class called EventHandler in order to support this, due to the limitations of Python being interpreted instead of compiled. While not quite as refined as it could be, this implementation is intended to support arbitrary events, so more functionality can be added to the program at a later date.

The color editing logic takes place within the ColorEditor.py module, which enables logic such as changing the color of the new frame, processing the color names database, and finding the closest color in the database given an arbitrary RGB value. Our original implementation relied purely on bit-masking the copy of the original frame, using the bitwise_not(), bitwise_and(), and inRange() methods provided by cv2 to try and layer enough edits to produce the desired result. After some iteration, this didn’t support more than one color being replaced at a time. Our final implementation still makes use of inRange(), however we make use of python list comprehension to go through every pixel and change its value.

We decided to determine the colors of our elements based on the color names supported by tkinter. This required a database, imported into our project, read, stored, and processed at runtime. A challenge arose from a need to match any arbitrary RGB value, to the database of known colors. Our idea was to compare the user RGB value to every database RGB value and accept the database RGB value/name with the lowest difference. After some research, we found the appropriate way to get the true difference between RGB values is to treat each of them as a vector and use vector subtraction to find the true difference. This method was successful in finding the true closest RGB database value/name given an arbitrary user RGB value. This also allowed us to display the true name of the current user RGB value in our GUI, providing relevant information to the user.

Finally, a crucial part of our implementation was the decision to switch from the queue python data structure to the collections.deque python data structure. When loading our program onto the Raspberry Pi, we noticed a significant reduction in performance, due mostly to our video producing thread creating more frames than the rest of our program could change and render to the UI, rendering a smooth but sluggish video. This prompted us to use deque to try and solve two problems at once; An online search suggested that deque’s have faster read and write times than Queue, which would reduce bottlenecked reading in our DisplayGUI and ColorEditor modules. Second, using the deque allowed us to artificially limit the number of frames our RealtimeVideo module was sending to our DisplayGUI and ColorEditor modules. This stopped frames piling up in the deque, eliminating the lagging effect, while only producing as many frames as we could consume, leading to a choppier but much more responsive video frame. The hardware of the raspberry pi required us to make compromises on our implementation and inspired a creative and functional solution to allow for a satisfactory product.

Conclusion

In conclusion, our Raspberry Pi project successfully demonstrated the feasibility of real time video feed analysis and color customization using a simple hardware setup. Despite facing some hardware limitations, we were able to create a functional and user friendly program that does what we set out to do. All of the goals we initially planned to complete were successfully implemented into Real Time Canvas, consisting of a live feed, analysis of said feed to display the color palette, and the ability to update those

colors and see a result in real time. In doing so we learned a lot about developing for Raspberry Pi, tkinter, GUI design, K-Means Clustering, OpenCV, python data structures, optimization, and much more. If we ever wanted to revisit the project, there's a number of additions that could be added to Real Time Canvas with very little effort due to the way frames are handled, such as adding support for color analysis and color editing on uploaded pictures or video files. Overall, we are very happy with how Real Time Canvas turned out, and are excited to develop more and more projects in the future as we continue our computer science careers.

Works Cited

A., M. Rake Linggar. "Finding Most Common Colors in Python." *Medium*, Towards Data Science, 9 Oct. 2020, https://towardsdatascience.com/finding-most-common-colors-in-python-47ea0767a06a.

Arvai, Kevin. "K-Means Clustering in Python: A Practical Guide." *Real Python*, Real Python, 30 Jan. 2023, https://realpython.com/k-means-clustering-python/.

Color Palette Cinema [@colorpalette.cinema]. Apocalypse Now Color Palette. *Instagram*, 12 Dec. 2022, https://www.instagram.com/p/CmErz2woth-/.

"Five Ways to Run a Program on Your Raspberry Pi at Startup." *Dexter Industries*, 2023, https://www.dexterindustries.com/howto/run-a-program-on-your-raspberry-pi-at-startup/.

"How Can I Prevent a Window from Being Resized with Tkinter." *Tutorials Point*, https://www.tutorialspoint.com/how-can-i-prevent-a-window-from-being-resized-with-tkinter.

"How to Get the Screen Size in Tkinter." *Tutorials Point*, https://www.tutorialspoint.com/how-to-get-the-screen-size-in-tkinter#:~:text=In%20order%20to%20get%20the,of%20the%20screen%20in%20pixels.

"Python - GUI Programming (TKINTER)." *Tutorials Point*, https://www.tutorialspoint.com/python/python_gui_programming.htm.

This article is written by plus2net.com team.https://www.plus2net.com. "Tkinter Rowconfigure & Columnconfigure to Assign Relative Weight to Rows and Columns Width & Height." *Python Tkinter Rowconfigure & Columnconfigure Using Weight*, plus2net.Com, 5 Feb. 2019, https://www.plus2net.com/python/tkinter-rowconfigure.php.