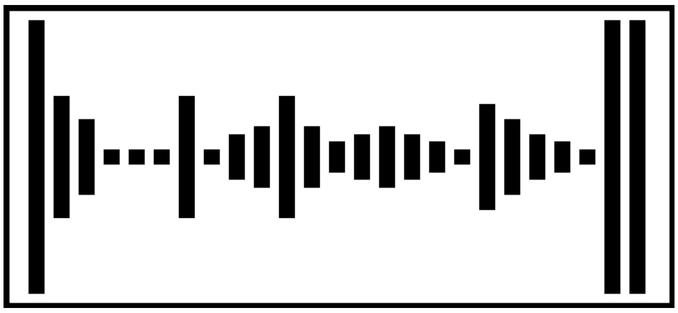
Embedded YouTube URL QR Codes

Joseph DiNiso[#1]

[#]*College of Engineering, Virginia Polytechnic Institute and State University*
Address Including Country Name
[1]`Joseph diNiso @josephdiniso@vt.edu`
Blacksburg, VA 24061 United States

Project GitHub Link: https://github.com/josephdiniso/YouQR

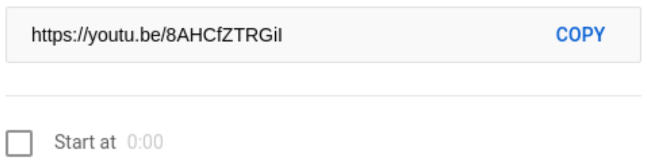


***Abstract*— Using digital image processing techniques such as image thresholding, contour calculation, and box detection, we developed a system of turning an embedded *YouTube* URL into a QR-like code that can be printed and scanned to open a browser window with the respective video.**

***Keywords*— Digital Image Processing, Object detection, QR Codes, Image scanning, box detection**

I. Introduction

The goal of our project was to develop a means of converting a YouTube URL to a format that would be easily scannable for a computer webcam. This would be used in a similar manner to *Spotify's* embedded URLs in which a sequence of bars can be scanned to open a song [1].



https://youtu.be/8AHCfZTRGiI
COPY
Start at 0:00

Example embedded YouTube URL

Example Spotify Code

II. Methods

The project had three main parts that needed to be implemented.

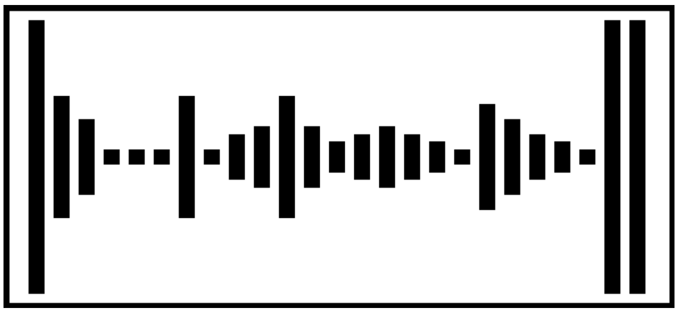
A. URL to Octal Representation

Firstly, a system of converting an embedded YouTube URL to a format of bars would require some code that could translate this. Upon researching the methods of the URLs, we came to the conclusion that YouTube uses a base-64 system, in which there are 64 unique characters used to represent a URL [2]. These characters range from A-Z, a-z, 0-9, and +, /. They each represent an integer from 0-63, (hence the name base 64). By converting a base-64 value to two octal values (base-8), we can represent a string of values by vertical bars that are in increments from size 1-8. For example, the base64 value 's' has an integer value of *44*, which is equal to *54* in octal representation. In other words, to represent the character 's' in a YouTube URL, we need two bars that are of heights *5* and *4*. We chose a base-8 system of representing the heights of the bars because we felt that it was a fair tradeoff between being able to represent a base64 character in just two bars and also being able to deal with a fairly large margin of difference in bar heights. If we had chosen a base-16 system, the difference in height between a bar of size 1 and size 2 would oftentimes be too small to see in a digital image, and the results would quickly get skewed. Because we are using a unique URL, any difference in values would lead to a completely different URL.

B. Code Generation

Now with a system of conversion from URL to bars in place, the program would need to be able to generate an image that represents the URL. In order to ensure that the image is oriented correctly, one vertical bar spanning almost the entire height of the enclosed code is displayed to the left, and two to the right of the code. This allows for the program to know if a code is oriented in the correct direction and can flip accordingly. As well, these bars are used as a metric to measure the ratio of height of the other bars.

In between the one bar on the left and two bars on the right are the data stored from the converted URL. Because the bars range in value from 1-8, each respective height is calculated as a percentage of the total height of the code. For example, a bar with the value of 1 would be given a height of 5% of the total height of the code, a bar with the value of 8 would be given a height of 40% of the total height of the code. This ratio will be shown to be important later when detecting the image. Shown in the following image is an example of a code generated from the song "Hurt" by Johnny Cash.


Generated code for "Hurt" by Johnny Cash

This code holds all of the information needed to open a link to a YouTube URL in the relative heights of the middle bars.

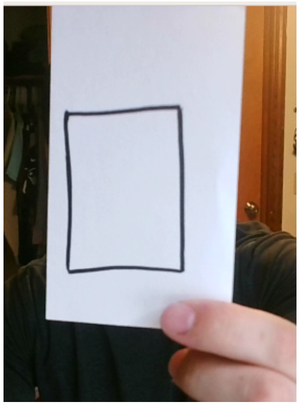
C. Code Detection

The part of the project with the most importance is the detection stage. In this stage a generated "QR" code of vertical bars must be detected by the program and converted back to its respective base64 URL representation.

This program begins by enabling the user's webcam and showing the webcam input to the computer. Each frame of the webcam input is converted to grayscale and then thresholded using OpenCV's adaptive thresholding function. A traditional thresholding function works by setting all pixels that are not within a specified range to 0, and sets all pixels within that specified range to 1 (generating a binary image). Adaptive thresholding, however, works better for real world scenarios as it considers different lighting in different parts of the image. This feature is extremely useful for removing glare and reflection from the image.

This adaptive thresholding function is a function written in C++ with bindings in Python that converts a grayscale image into a binary one given a few parameters - mainly the thresholding method and the output type. The reason we decided to use built-in OpenCV image processing functions instead of implementing our own is because Python by nature is a very slow programming language. Because it is in fact a scripting language, code will execute much slower compared to a language like C or C++ where OpenCV was written. Since speed was a major concern for us, as we are dealing with live video input, we had to use the optimized functions.

After converting to a binary image, we then used OpenCV's findContours function, which returns a list of point coordinates of potential shapes in the image. The findContours function works by generating a *contour*, which is simply a list of points in an arc of similar color or intensity. Because we are using a binary thresholded image, arcs should be fairly easy to generate as they consist simply of 1 or 0. Since the only shape we are looking for is an enclosed rectangle around the "QR" code we use another built-in OpenCV function approxPolyDP() This function determines the number of sides of a contour in the list of contours generated from the findContours function. If the number of sides that approxPolyDP() is equal to four (representing a rectangle-like shape), we then check if the aspect ratio of the resultant rectangle is suitable (within the range of (1.8, 2.2)). If it is suitable, a crop is then made consisting solely of the pixels enclosed by the rectangle and this crop is passed to another detection function which will work on the inner bars. This aspect ratio check can be shown by the following image, in which a drawn rectangle does not get shown because it does not fit the correct width to height ratio.



This second detection function performs a similar rectangle detection as the first using very similar methods, however this time it is looking for the enclosed vertical bars which represent information about the URL. Measuring the ratio of each bar to the height of the rectangle, approximations of their octal values are then calculated. These octal representations are then passed into a function which takes pairs of octal values and converts them back into base-64 values.

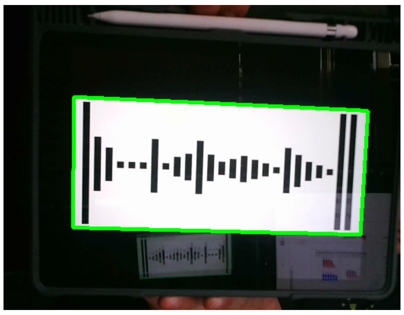
If valid base-64 values are calculated, the program then ends its webcam loop, and opens a window of the YouTube URL that it calculated.

III. Experimental results

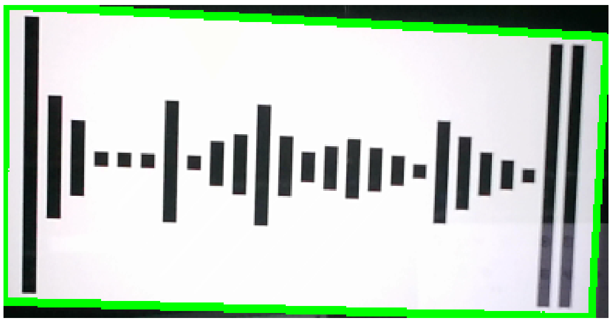
Testing of the system consists of two parts - generation and detection. Generation was tested simply by supplying the program various different URLs and ensuring that the output could be converted immediately back into its original data - i.e. no loss of data from the conversion from base-64 to octal and then back.

The second part - detection, required a much more intuitive approach, because the program is intended to work on live data from a webcam feed, testing was mainly done simply by giving the webcam various different QR codes and ensuring that the program detects them well.

An example of an input to the webcam feed is shown below, in which I am holding an iPad with an image of generated QR code.

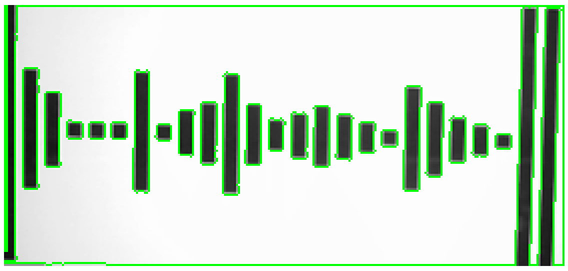


The crop made from this image can then be seen below.



This crop is then used in the aforementioned methods to generate the code “8AHCfZTRGiI” which is the link to the song “Hurt”.

Shown here are the respective contours detected for the given image. Note that the program also detects the outlying rectangle as a contour - this problem is avoided by ensuring a contour rectangle’s area is under a certain percentage of total image size.



Upon testing this program several hundred times we have come to the following conclusions about the effectiveness of it:

Firstly, noise such as glare or reflection (which tend to be much more prevalent on a digital screen) will greatly reduce the effectiveness of the program. This is likely because reflections of the screen can skew the program’s detection of the bounding rectangle, and skew the program's perception of the height of each bar. Perhaps this noise can be further reduced with better thresholding and noise reduction.

Secondly, in the current build of the program, detections of the rectangle at an angle can lead to issues with the ratio of height, giving skewed results as the ratio of height in the left of the crop may not be the same as in the right. In future iterations adding functionality to rotate the rectangle so that it has straight edges would be beneficial in fixing this.

This detection program works best when there is little glare, not much reflection, and a well-aligned image. If given those three factors, the program is effective almost 100% of the time - a partial reason for this is that the program will only exit once it has successfully opened the correct URL.

IV. *Discussion*

This project required us to use a plethora of different computer vision and digital image processing techniques to create a working program.

As discussed before, the majority of the computer vision techniques applied for this project were not developed by us, but instead used built in OpenCV functions, such as image thresholding, contour detection, polygon corner detection, and more. In future iterations of the project we would like to build off on this by implementing the algorithms ourselves by creating new C++ bindings and extending Python's abilities as such. By writing the digital imaging functions in C++ but utilizing them in Python we can ensure that our code remains simple, readable, and intuitive while also capitalizing on the speed of C++.

Overall we are very content with the results of our project. We feel that we built a program that works exceptionally under the desired circumstances. However, another addition to be added in future iterations is better usage of corner detection on rotated rectangles. By detecting four corner points instead of just one corner point and then specifying the width and height, we can ensure that images that are captured at an angle can still be used to open a URL.

IV. Conclusions

This project helped to solidify our knowledge of the digital image processing domain and begin to understand both the field and its applications better by creating a working program with it. Our goals of creating a program that can effectively generate and detect our own version of a "QR" code were certainly succeeded, and we feel that we outdid our expectations of the efficacy of the project. While our application was applied only to YouTube URLs, this technique can certainly be applied to any platform's embedded URLs - such as tinyURL, Twitter, etc. We would perhaps also be interested in the future in adapting this to other applications, and perhaps making it compatible with other number systems, not just base64 to ensure its code reusability.

V. Contributions

Walter and I split the work up based on our strengths. There were a few sections of the code that needed to be completed: the conversion from base64 to octal and then to bars, the generation of the QR image, and the detection of the QR image. I had completed the conversion algorithm which takes in a string of characters and returns a list of octal integers. Walter did the majority of the bar generator, in which given a string of b64 characters, will save an image of a QR code to the user's directory. For the detection algorithm, we split the work almost 50-50, in which Walter worked on the bar detection and I worked on the outer box detection as well as getting the code to work properly with a webcam. In all, I would say that we split the workload almost evenly 50-50.

References

[1] Pocket-lint, "What are Spotify Codes and how to use them?," Pocket, 26-Jan-2021. [Online]. Available: https://www.pocket-lint.com/apps/news/spotify/140964-what-are-spotify-codes-and-how-to-use-them. [Accessed: 04-May-2021].

[2] "What type of ID does YouTube use for their videos?," Stack Overflow, 01-Jan-1958. [Online]. Available: https://stackoverflow.com/questions/830596/what-type-of-id-does-youtube-use-for-their-videos#:~:text=Normal%20Base64%20is%20%5BA%2DzA,%2D%20and%20_%20are%20substituted%20respectively. [Accessed: 04-May-2021].