Brian Cornet
DSC 498
Dr. Donghui Yan
12/8/2020

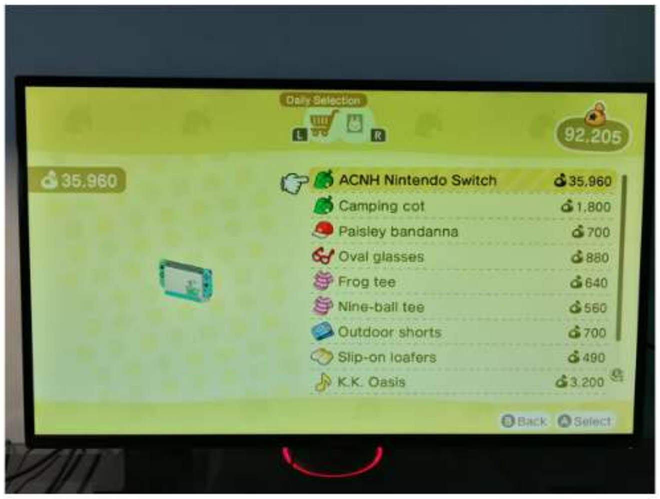
Capstone Project I: Final Report

Working project title: Raw Image Scanner (RIS)

Introduction

This project aims to create a data collection method for camera images and videos of digital displays. Specifically, it is intended for a user to be able to provide an input (an image, video, etc.) of a source device's display (a television, monitor, phone screen, etc.) that contains some kind of quantitative data collection (a list or table of text strings, a collection of images, etc.) to collect and organize that data. The intent is to provide a quick means of copying structured information from one device to another for instances where accessing the raw data directly is inconvenient or prohibited. Typically, this restriction is enforced by design for closed platforms such as video game consoles (Nintendo Switch, Sony PlayStation 4, etc.) and iOS devices from Apple Inc., file formats that may be encrypted, or data streams that may not be recorded locally. Furthermore, many users lack the technical skills to access and manipulate arbitrary data structures directly and instead rely on existing software appropriate to the file format to do so. In all instances, the data itself is not private to the user, but any collection will have to be done by hand.

For example, a user may be playing a video game with the goal of collecting over 10,000 distinct items. The game provides a text list of every item obtained thus far (displaying 10 items at a time) but offers no hints towards which items are missing or how many items remain (i.e. a list of distinct items $N =$ {"Apple", "Ball", "Car"} with no indications as to whether "Apricot" or "Xylophone" are in $\overline{N}$ or how large $|N \cup \overline{N}|$ may be). Suppose the user wants to make a checklist of which items they currently had and cross-reference it with a list of known items obtained online to determine which items they're missing. While the data exists in the internal memory as a sequence of binary values (0 = missing, 1 = obtained), the internal memory itself is inaccessible to the user. The user also has thousands of items, many of which have similar names ("Wooden End Table", "Wooden Low Table"), are difficult to spell or type ("Myllokunmingia", "Übel/Equité"), or are simply long ("Archive Tower Giant Door Key", "Imperial Army Identification Tag"), making compilation by hand tedious and prone to errors. The user can easily record a video of themselves scrolling through the entire list with their personal smartphone, creating a digital basis for their data. By using optical character recognition (OCR) methods, a program could examine the frames of the video for text strings representative of every item the user has obtained, transforming their original digital basis into a more concise and appropriate data structure for the cross-reference operation.



ACNH Nintendo Switch	35,960
Camping cot	1,800
Paisley bandanna	700
Oval glasses	880
Frog tee	640
Nine-ball tee	560
Outdoor shorts	700
Slip-on loafers	490
K.K. Oasis	3,200

$\longrightarrow$

ACNH Nintendo Switch
Camping cot
Paisley bandanna
Oval glasses
Frog tee
Nine-ball tee
Outdoor shorts
Slip-on loafers
K.K. Oasis

Figure 1. Converting an image of an item list into a set of strings. Image from Animal Crossing: New Horizons, Nintendo 2020.

Text-less images can also be used for comparison in a “spot the differences” problem. For example, a user is playing a video game with the goal of visiting every room in a large castle at least once. The game provides a representation of the castle’s map that shows a visited room’s position and hides unvisited rooms, as well as a percentage of the total rooms visited. Again, the digital data would represent each room with a binary value (0 = unvisited, 1 = visited) but cannot be accessed by the user. A clean image of the completed map is available online. The user knows they have a small number of unvisited rooms remaining but is unsure of which ones and exactly how many. Rather than try to detect which rooms are unvisited by carefully examining the differences between the image, the user could take a picture of their current map. This picture could then be rotated and skewed to make the orientation of the completed map, resized to fit its dimensions, and cropped to match its positions before both maps are converted into binary matrices representing each room. Assuming the images are both accurate, a difference between any two elements at a given coordinate would indicate the locations of a missing room, which could then be highlighted on either map.

A screen capture from any digital display would be preferable for an accurate interpretation over a photograph of a screen. Generally, most captured text and graphics will be static, smooth, properly aligned, correctly sized, framed, and free of environmental noise or errors. Camera images

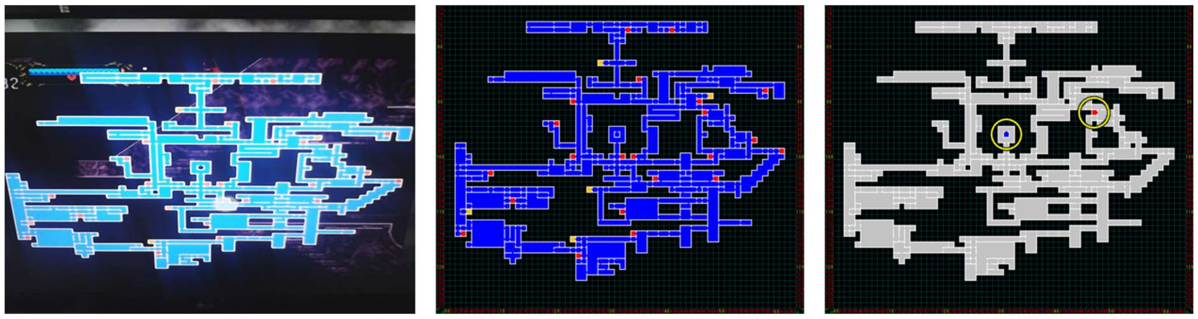


Figure 2. Detecting the differences between two images. Image from Castlevania: Symphony of the Night, Konami 1997.

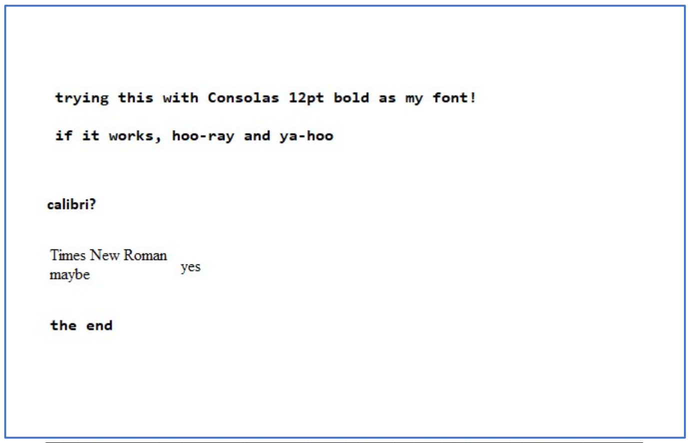
meanwhile are subject to any number of errors caused by the angle of the lens relative to the screen, the lighting of the environment, objects outside the display's boundaries (including the display's frame), unsteady positioning when held by hand, detection of interlacing between frames and display pixels (moiré), or filters and corrections applied by the camera's software. However, one of the primary objectives of this software is to improve the convenience of collecting data, and not all users will have access to screen capturing and transfer methods. Comparatively, mobile phones equipped with cameras are extremely popular and convenient. Ignoring compatibility with camera images should thus be considered insufficient.

The bulk of programming work in this project will be conducted in Python due to the abundance of machine learning libraries associated with image recognition and correction. OCR for text-based images is achieved through Tesseract 4 and implemented in Python using the PyTesseract library. Per Tesseract's documentation, retraining methods are not explored as they are considered unlikely to help improve output accuracy. Instead, the image itself is transformed in various ways to improve quality and allow Tesseract to better "see" the appropriate text. Image correction methods are mandatory in the correction of camera images regardless and will be a primary focus of this project.

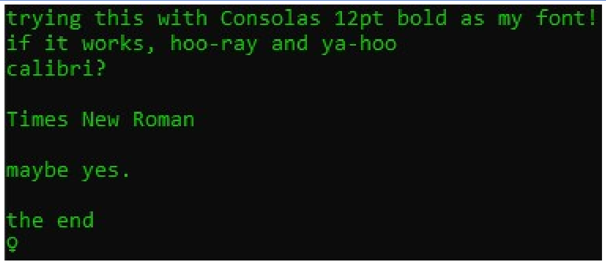
Initial OCR Experimentation

Though this software is intended to be able to compare images, its first goal is to successfully implement OCR methods for accurate text collection. Direct image comparisons should be relatively simple to implement following the completion of the OCR process given the overlap in image correction methods needed. As a result, the development of direct image comparison methods has yet to begin.

Initial OCR experiments were conducted on several images of different font. First, a control image "test.bmp" which contains black text on a white background generated in Microsoft Paint. The Consolas (bold), Calibri (bold), and Times New Roman (normal) fonts were printed in several locations with clear lines and spacing between each line. Figure 3 shows the original image next to the recognized text. Note that the result was slightly imperfect: an additional period or "." was erroneously detected after the word "yes", which was not detected until after adding the "yes" to the image. The "calibri?" line was also considered to be part of the Consolas block above it despite being closer to the Times New Roman block below it. Both errors suggest that Tesseract prefers text to be consistently aligned horizontally and vertically when presented with text spanning multiple lines.



```
trying this with Consolas 12pt bold as my font!

if it works, hoo-ray and ya-hoo


calibri?

Times New Roman
maybe             yes

the end
```

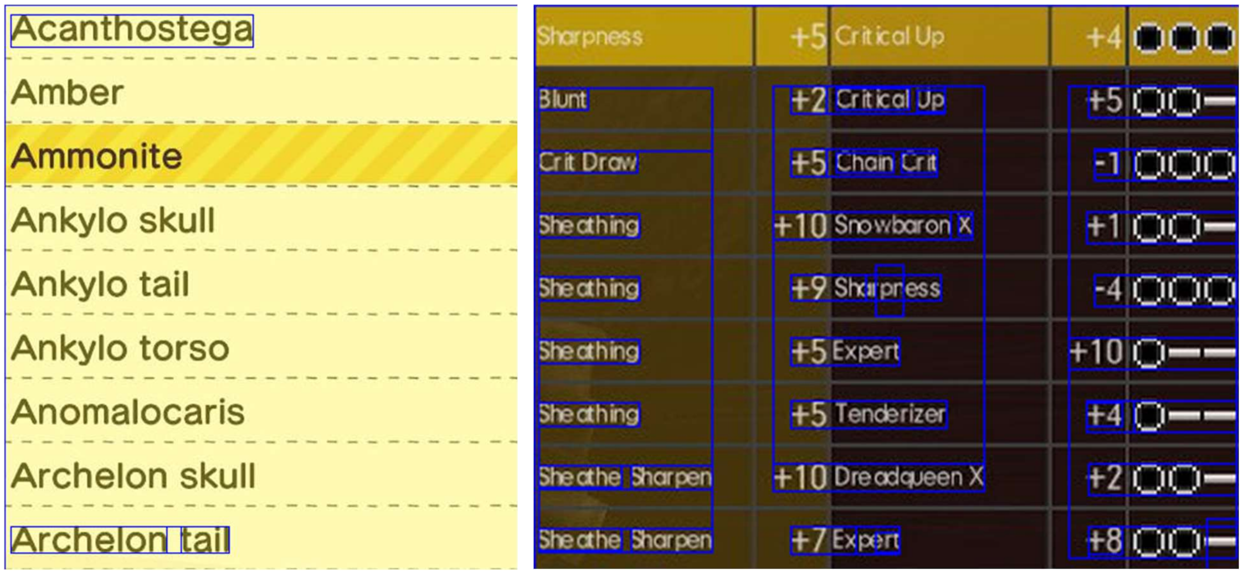


```
trying this with Consolas 12pt bold as my font!
if it works, hoo-ray and ya-hoo
calibri?

Times New Roman

maybe yes.

the end
♀
```

Figure 3. "test.bmp" (with border added for visibility) and results.

Next was a test using a list from *Animal Crossing: New Horizons* (Nintendo 2020). The font in this image is very simple, perfectly aligned, and features a green font on a yellow background (except for a highlighted row which has brown text against a yellow and green striped background). The initial attempt produced little useful text: "@YPacanthostesa" for an icon of an animal skull and bone next to the word "Acanthostega" for the intended result area. By cropping the image to just the relevant text section, the results were able to correctly read the first and last entries "Acanthostega" and "Archelon tail", but nothing else. Selecting a single item in the middle of this list correctly produced the intended result "Anomalocaris", indicating that the issue was likely caused by the highlight bar. This shows the potential of cropping blocks of text to improve accuracy directly.

The final test used a list from *Monster Hunter Generations Ultimate* (Capcom 2018). The relevant text in this image is white on a dark background with some transparency showing background details. The font is fairly small with some overlap between characters with some artifacts around characters due to lossy compression. The first and third columns contain left-aligned word strings, the second and fourth columns contain right-aligned number strings with a preceding plus or minus sign, and the fifth column contains three symbols that can be circles or hyphens. The first two columns are also highlighted with a light brown background, the first row is highlighted with a green background, and the intersection is a slightly brighter green. Unlike the previous test, the accuracy of this test failed to reach 100% for a single row due to the difficulty in perceiving certain words (“Snowbaron” was interpreted as “Srowboron”), the mathematical symbols next to the numbers (“+1” interpreted as either “S” or “4”), and the inability to comprehend the circles and dashes in the last column as special symbols. This test demonstrates the importance of image correction even for captured images, as well as the need for uncommon symbol comprehension or interpretations of specific shapes or patterns.



Acanthostega
Amber
Ammonite
Ankylo skull
Ankylo tail
Ankylo torso
Anomalocaris
Archelon skull
Archelon tail

Sharpness	+5	Critical Up	+4	●●●
Blunt	+2	Critical Up	+5	○○—
Crit Draw	+5	Chain Crit	-1	○○○
Sheathing	+10	Snowbaron X	+1	○○—
Sheathing	+9	Sharpness	-4	○○○
Sheathing	+5	Expert	+10	○——
Sheathing	+5	Tenderizer	+4	○——
Sheathe Sharpen	+10	Dreadqueen X	+2	○○—
Sheathe Sharpen	+7	Expert	+8	○○—

Figure 4. “acnh.png” (left) and “mhgu.png” (right) cropped with highlighting over text areas recognized.

Image Correction Methods

In addition to cropping, the accuracy of Tesseract’s output can be improved through other methods of image correction. Several of the methods have been explored and implemented in the project’s environment, though some remain to be fully explored. Combining several methods in varying orders has shown to produce different outputs, so an analysis of permutations may be necessary to improve output accuracy and time complexity.

Inversion

This is a simple process where the entire image is inverted -- black becomes white, white becomes black, and so on. If every pixel is represented by a string of bytes, then this can be achieved through a binary XOR operation on each individual byte with the constant bit string 11111111 (255 in decimal, 0xFF in hex). Tesseract’s documentation specifies a preference for dark text on light backgrounds as seen in “test.bmp” and “acnh.png” in figures 3 and 4 respectively. This should improve the results of an image with light text on a dark background such as “mhgu.png” in figure 4.

Experimentations with “test.bmp” and “acnh.png” showed that their inverted images produced results with 0% total accuracy, whereas an inverted “mhgu.png” showed a significant improvement in accuracy for most words with a few instances of reduced accuracy for single characters at specific positions.

Rescaling

Based on experiments conducted by “Willus Dotkom” accessible through Tesseract’s Improve Quality page, Tesseract 4 works best when analyzing text where the height of capital letters (in pixels) is within a range of approximately 20 to 36 pixels depending on the font. Some fonts such as Helvetica-Narrow were observed to have errors within this range. To account for the possibility of unusual fonts, the program is designed to rescale the image to multiple pixel sizes and evaluate each font size independently. It may be worth investigating retraining methods for particularly unusual fonts as recommended by the Tesseract documentation. PyTesseract’s image_to_boxes() function provides the coordinates for each individual character’s start and end positions for each dimension. Finding the height for capital letters should be trivial except in cases where no capital letters are detected. Inclusivity for other tall characters such as “d” or “8”, verification of the first characters in words, and verification of characters that look similar in upper- and lower-case (e.g. “X” and “x”) may be warranted.

Binarization

This process assigns each pixel to be either black or white, providing a clear contrast between edges and making shape recognition easier. Tesseract natively performs this through Otsu’s method but recommends other methods for uneven background colors from the OpenCV and scikit-image modules.

One method explored was to separate the image into bitplanes by splitting the color values of each pixel into its own binary image. Input images are assumed to have a 24-bit color map, with 8 bits (1 byte) assigned to red, green, and blue color brightness respectively. Describing the image in an environment such as MATLAB or Python’s Numpy module typically results in a $M \times N \times B$ tensor of unsigned 8-bit integers representative of $M \times N$ pixels and B bytes. Reshaping these image tensors into a shape of $M \times N \times 8B$ binary values can be done with AND operations over each $M \times N$ matrix in B. The resulting images are bitplanes and can be displayed or examined individually. Note that the bit length of an image is irrelevant; this method is applicable to other formats such as 8-bit grayscale or 32-bit color images with transparency. Any n-bit color values can be split into n separate bitplanes. In most cases, the most significant bit (0x80) will provide most if not all of the relevant text while less significant bits instead contribute to noise. However, there may be cases where the detail provided from less significant bits may be relevant.

An experiment using bitplane decomposition on "acnh.png" demonstrates the effectiveness of this method in instances where certain text/background color combinations interfere with output accuracy. This image was originally unable to detect any words between the first and last lines due to the highlights included on the third line. By separating the image into bitplanes, the color changes applied to the highlighted row are conveniently ignored for the most significant red and green bits. Tesseract is then able to interpret the text with 100% accuracy from either bitplane. The results shown in figure 5 indicate the importance of the green color byte in this image as it had the highest average accuracy across its own subset of bitplanes. Meanwhile, the blue color byte is clearly identified as the cause of the original error seen in figure 4.

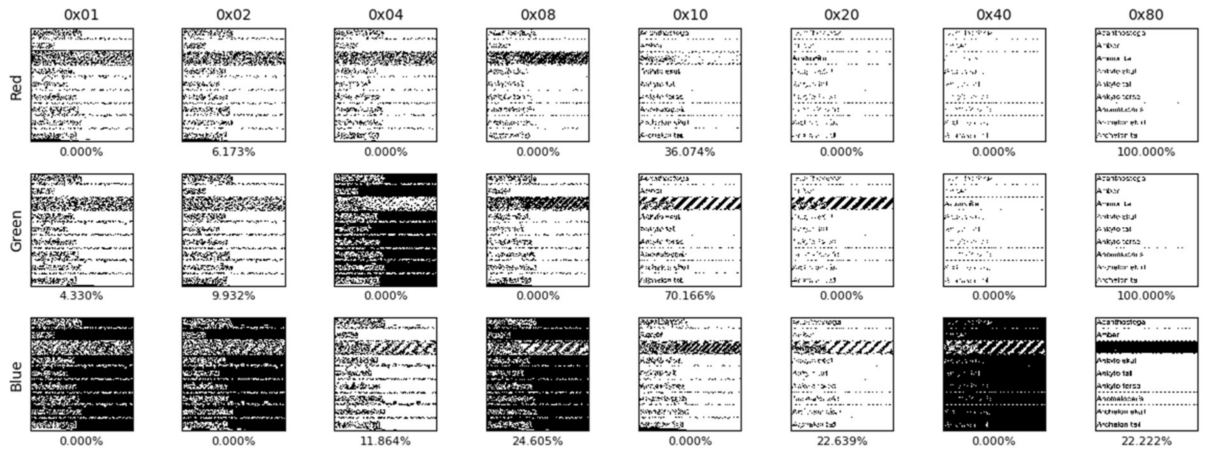


Figure 5. "acnh.png" separated into 24 bitplanes. Ordered least to most significant bits left to right.

Other methods

Several other methods are suggested by Tesseract's documentation. Some of these methods have been implemented through manual processes such as cropping but were not formally included through sophisticated module methods. These include the following:

- Noise removal, where random variations in brightness are removed.
- Dilation and erosion, where edges of shapes are made thicker or thinner to increase visibility.
- Rotation, where a skewed image is aligned vertically and horizontally.
- Border removal, where noisy backgrounds are cropped to prevent being interpreted as characters.
- Border addition, where a solid border is added to help Tesseract designate a text area.

Proper use of these techniques will be essential to the success of OCR in camera images and are thus considered mandatory for this project's future.

Conclusion

Converting images to simpler discrete data structures for comparison or storage should be possible with the methods described in this report. Tesseract 4 serves as a reliable platform for collecting text from images and will remain as the core platform for OCR implementation. The overall potential of this project relies on various image correction methods to produce reliable interpretations of source data, particularly for camera images. Several such methods have been investigated in limited contexts; other remaining methods will be explored in the future. Other considerations for future developments include the direct image comparison method and an implementation of machine learning algorithms to improve the accuracy of OCR with unusual fonts. Additional concepts not mentioned here may also be introduced as needed.

Brian Cornet
DSC 499
Dr. Donghui Yan, Dr. Scott Field
4/7/21

Capstone Project II: Final Report

Project Title: Optical Character Recognition and Correction for Images of Digital Displays

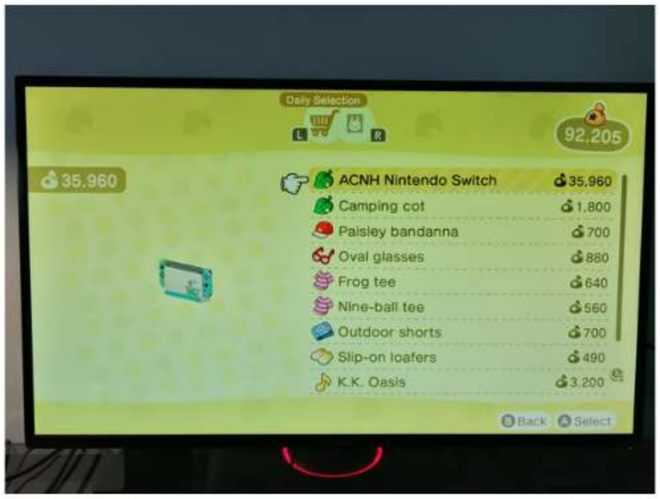
Abstract

Data structures such as tables and lists are commonly presented through graphical user interfaces (GUIs). In many cases, accessing the data directly is inconvenient or prohibited by the platform itself (Apple iOS, Nintendo consoles, etc.) or unavailable locally as with video streaming. Notably, the data is not private to the user, but collection must be performed by hand. This project introduces a program for Optical Character Recognition (OCR) using Tesseract 4 and Python to convert data displayed in images or videos into discrete structures. Image transformations such as binarization, scaling, skewing, edge detection, and denoising enable compatibility with elaborate GUIs and photographs of digital displays. Predictive methods using Levenshtein and character-shape distances provide text correction to significantly improve output accuracy. Error analysis is also conducted for individual correction methods using several sample images, dictionaries, and fonts.

Introduction

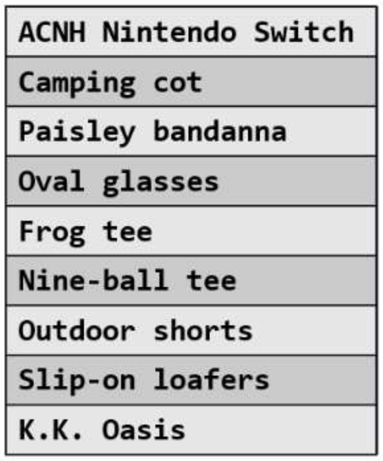
This project aimed to create a data collection method for camera images and videos of digital displays. Specifically, it intended for a user to be able to provide an input (an image, video, etc.) of a source device's display (a television, monitor, phone screen, etc.) that contains some kind of quantitative data collection (a list or table of text strings, a collection of images, etc.) to collect and organize that data. The intent was to provide a quick means of copying structured information from one device to another for instances where accessing the raw data directly is inconvenient or prohibited. Typically, this restriction is enforced by design for closed platforms such as video game consoles (Nintendo Switch, Sony PlayStation, etc.) and iOS devices from Apple Inc., file formats that may be encrypted, or data streams that may not be recorded locally. Furthermore, many users lack the technical skills to access and manipulate arbitrary data structures directly and instead rely on existing software appropriate to the file format to do so. In all instances, the data itself is not private to the user, but any collection will have to be done by hand.

For example, a user may be playing a video game with the goal of collecting over 10,000 distinct items. The game provides a text list of every item obtained thus far (displaying 10 items at a time) but offers no hints towards which items are missing or how many items remain (i.e. a list of distinct items $N = \{$“Apple”, “Ball”, “Car”$\}$ with no indications as to whether “Apricot” or “Xylophone” are in $\overline{N}$ or how large $|N \cup \overline{N}|$ may be). Suppose the user wants to make a checklist of which items they currently had and cross-reference it with a list of known items obtained online to determine which items they’re missing. While the data exists in the internal memory as a sequence of binary values (0 = missing, 1 = obtained), the internal memory itself is inaccessible to the user. The user also has thousands of items, many of which have similar names (“Wooden End Table”, “Wooden Low Table”), are difficult to spell or type (“Myllokunmingia”, “Übel/Equité“), or are simply long (“Archive Tower Giant Door Key”, ”Imperial Army Identification Tag”), making compilation by hand tedious and prone to errors. Instead, the user could easily record a video of themselves scrolling through the entire list with their personal smartphone, creating a digital basis for their data. By using optical character recognition (OCR) methods, the program would examine the frames of the video for text strings representative of every item the user has obtained, transforming their original digital basis into a more concise and appropriate data structure for the cross-reference operation.



ACNH Nintendo Switch	35,960
Camping cot	1,800
Paisley bandanna	700
Oval glasses	880
Frog tee	640
Nine-ball tee	560
Outdoor shorts	700
Slip-on loafers	490
K.K. Oasis	3,200

$\longrightarrow$



ACNH Nintendo Switch
Camping cot
Paisley bandanna
Oval glasses
Frog tee
Nine-ball tee
Outdoor shorts
Slip-on loafers
K.K. Oasis

Figure 1. Converting an image of an item list into a set of strings. Image from Animal Crossing: New Horizons, Nintendo 2020.

A screen capture from any digital display would be preferable for an accurate interpretation over a photograph of a screen. Generally, most captured text and graphics will be static, smooth, properly aligned, correctly sized, framed, and free of environmental noise or errors. Camera images meanwhile are subject to any number of errors caused by the angle of the lens relative to the screen, the lighting of the environment, objects outside the display's boundaries (including the display's frame), unsteady positioning when held by hand, detection of interlacing between frames and display pixels (moiré), or filters and corrections applied by the camera's software. However, one of the primary objectives of this software was to improve the convenience of collecting data, and not all users will have access to screen capturing and transfer methods. Comparatively, mobile phones equipped with cameras are extremely popular and convenient. Compatibility with camera images was thus deemed a major goal of the project.

The bulk of programming work in this project was conducted in Python due to the abundance of machine learning libraries associated with image recognition and correction. OCR for text-based images was achieved through Tesseract 4 and implemented in Python using the *PyTesseract* library. Per Tesseract's documentation, retraining methods were not explored as they are considered unlikely to help improve output accuracy. Instead, images themselves were transformed in various ways to improve quality and allow Tesseract to better "see" the appropriate text. Image correction methods are mandatory in the correction of camera images regardless and became the initial focus of this project. Several individual methods for image correction were employed with varying degrees of success, the majority of which saw implementation through OpenCV's *cv2* library in Python.

Partway through development, focus shifted towards implementing dictionary-based methods for text correction. OCR errors are commonplace, even when parsing text from flawless images. Assuming the user knows every possible character or string for a single block, a method for comparing the OCR result with a list of these possibilities to determine the best match could provide the most likely interpretation. Supporting this idea would be a method for recognizing screens by form, allowing the program to remember text region locations and their associated transformations and dictionaries for improved performance speed and accuracy.

It should be noted that this program does not take advantage of advanced machine learning techniques (specifically convolutional neural networks) beyond the implementation of Tesseract and OpenCV. While the efficacy of such methods may improve the output, a lack of formal training and experience with these methods until later in the project's lifespan was recognized as a limiting factor early in development. This project was meant to be a proof-of-concept – these methods were deliberately unexplored and left for any future works.

Image Correction Methods

OCR output relies on two specific tasks: recognizing the locations of text in an image (**location identification**) and correctly identifying the characters in these locations (**character identification**). Short of developing a new detection model or direct user intervention, image correction is notable for being the primary means of improving location identification. Subsequently, character identification improves when applying these methods, though some design decisions (unusual fonts, noisy backgrounds, etc.) typically limit the upper accuracy bounds. While Tesseract applies some of these methods to an individually scanned section, the platform's documentation recommends independently improving the quality of the image prior to scanning. Three such quality improvement methods were explored in this project: binarization, rotation, skewing, and noise removal. Other unexplored methods such as dilation, erosion, border addition, and border removal may be considered in future works.

Binarization

This process assigns each pixel to be either black or white, providing a clear contrast between edges and making shape recognition easier. Tesseract natively performs this through Otsu's method but recommends other methods for uneven background colors from the OpenCV and scikit-image modules. Note that OCR generally performs better when examining dark text on a light background; light text on dark backgrounds can be easily corrected by inverting the colors on an entire image or section.

One method explored was to separate the image into bitplanes by splitting the color values of each pixel into its own binary image (**bitplane separation**). Input images are assumed to have a 24-bit color map, with 8 bits (1 byte) assigned to red, green, and blue color brightness respectively. In many cases, the most significant bits define the most significant features such that scanning these sub-images may have better output than scanning the original image. Applying a grayscale filter to the image and collecting the most significant bit of the newly-created 8-bit color map provides an alternative method of evaluating these bits with less run time. These methods were very successful in images where the text and background colors are consistent, though eliminating less significant bit values managed to reduce some noise effects such as those introduced from camera images.

Another method is **thresholding**, where the value at each pixel is rounded to 0 or 255 based on a given brightness cutoff. Iteration over a range of cutoffs can be performed over a given image to determine a likely prediction. The output of each scan at each threshold is recorded, and the most commonly-reported output at each section is used as the prediction. Using this method in conjunction with bitplane separation and grayscale filtering can provide an incredibly large sample size of outputs to consider and use in predictions. However, the relatively long runtime for each scan encourages the use of a limited selection of possibilities. This can be found using a smaller set of sections within the image. For example, using the sample image “mhgu3.png”, the accuracies for each of the five values over a threshold range of 1 to 254 were collected by comparing the Levenshtein distance between the outputs and the actual values. The threshold range between 167 to 176 was the largest range with the highest accuracy between these values, and so it is used in evaluating “mhgu2.png” (from which the previous image originates).

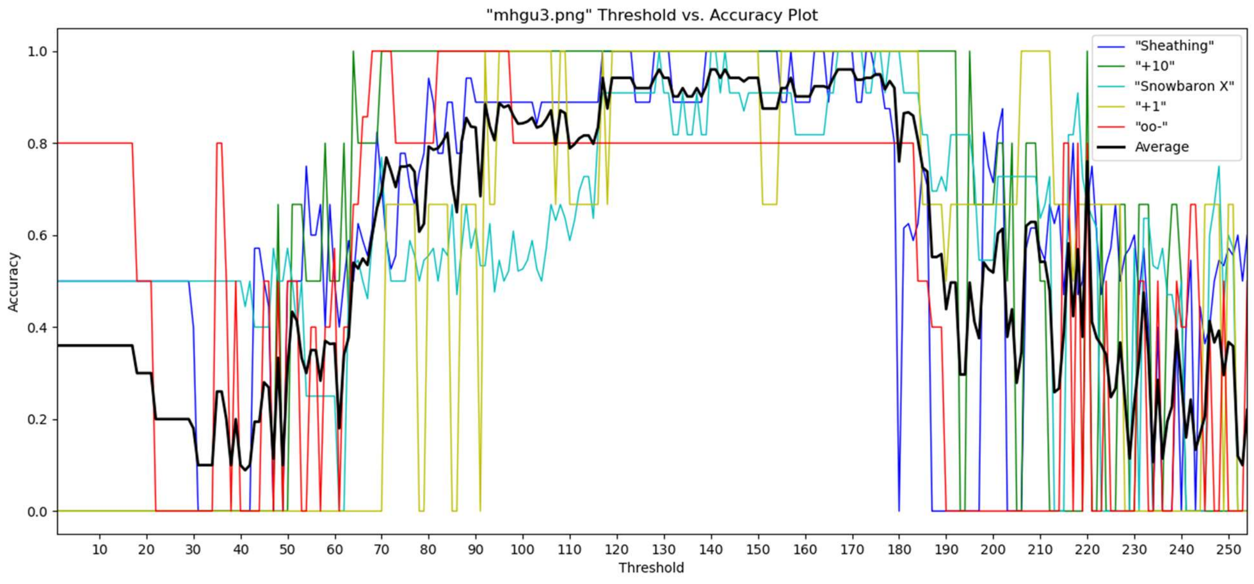


Figure 2. Evaluating OCR accuracy at threshold values from 1 to 254.

Rotation and Skewing

All camera images are taken at some angle relative to their display – no human should be expected to hold or mount a camera perfectly parallel to a screen. To account for this, rotation and skewing methods are needed to realign the image so that Tesseract can interpret the shapes of the characters correctly. Frequently, images where the region of interest (ROI) is only a small part of the screen will see more difficulty in performing these methods reliably. Cropping out the relevant section manually may be necessary in such cases.

One successful rotation and skewing method was **matching**, where similar features are detected between the camera image and a template image using the **Features from Accelerated Segment Test (FAST)**, **Binary Robust Independent Elementary Features (BRIEF)**, and **Oriented FAST and Rotated BRIEF (ORB)** algorithms. The locations of these features (as x,y coordinates) are used to create transformation matrices that rotate and skew the camera image to match the orientation of the template. This is usually successful when a clean template image is available (such as an empty table using the same design as the camera image) and neither image has too many or too few features. Given that these prerequisites may not always be realistic, other methods must be considered.

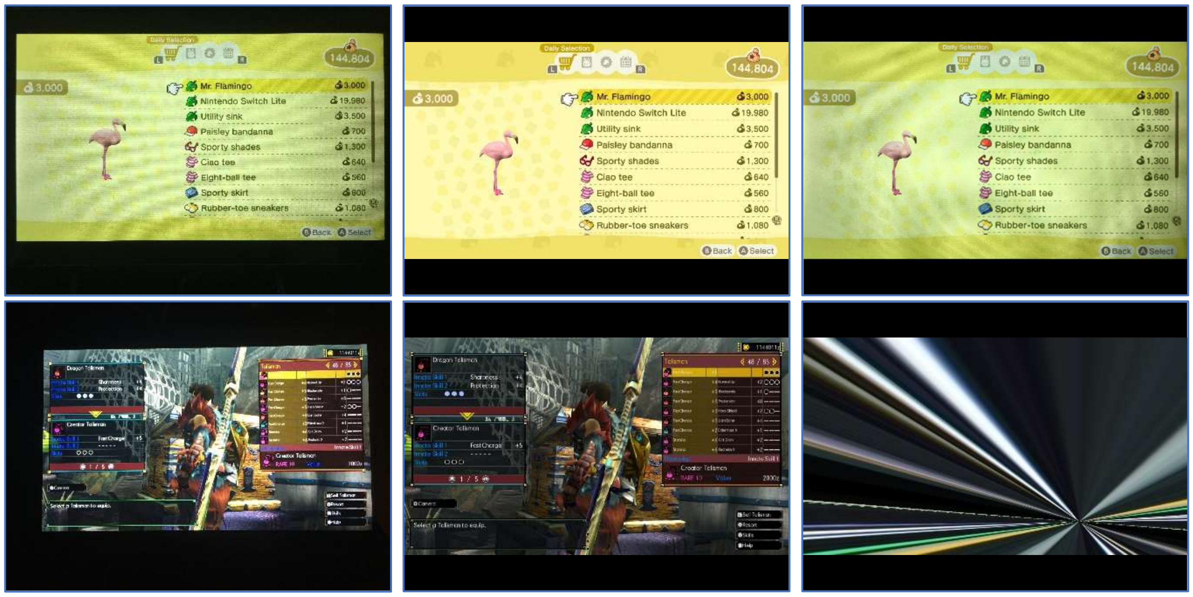


Figure 3. Matching attempts on images from Animal Crossing: New Horizons *(Nintendo, 2020) and* Monster Hunter Generations Ultimate *(Capcom, 2018).*

Another method involves using **Canny edge detection** and **Hough lines** to determine the location of the screen boundary or major lines that indicate the direction of the x and y axis on the original display. The former of these methods relies on a series of transformations: first, a **grayscale filtered** image is smoothed using a **Gaussian filter**, then **Sobel filters** are applied to determine horizontal and vertical derivatives which are used to find intensity gradients. **Lower bound cut-off suppression** is applied such that only pixels brighter than both of its adjacent pixels (adjacency direction based on the angle θ returned by the arctangent of the two derivatives) retain their gradient value. **Double thresholding**, where values are mapped into “suppressed” (0), “weak” (0 <= *n* <= 255), and “strong” (255) categories, is applied based on cutoff values relative to the brightest gradient. Finally, edge tracking by **hysteresis** maps any weak pixels to strong if any adjacent pixels are strong (8-directional) and suppresses it otherwise. An additional method to filter out interior Canny edges can also be applied, though this assumes no irrelevant exterior Canny edges are detected (typically objects and regions outside of the display device such as walls, furniture, or the user’s body). Hough lines can then be predicted, allowing the **Harris** or **Shi-Tomashi corner detection** methods to calculate the parameters used in the rotation and skewing process. Note that radial distortion from certain camera lens can introduce curving which affects the accuracy of the Hough line prediction; methods for removing this distortion were not fully realized at this time.

Noise Removal

Image noise (random variation in color and brightness) is introduced when taking any camera image and should be eliminated whenever possible. Additionally, some interfaces feature “noisy” fonts or backgrounds, especially those using transparency over some other image layer. Camera images of digital displays also usually introduce moiré patterns from overlaying the camera and display’s scan lines, which may be trickier to eliminate in particularly heavy regions. Typically, the simplest way to remove noise is to use blurring filters (such as Gaussian filters) and binarization methods described previously. Taking pictures with the camera rotated 30° and parallel to the screen with the ROI in the center of the image should ideally minimize moiré patterns. High brightness on the display, low environmental lighting, and a clean screen free from scratches, smudges, dust, etc. are also recommended for these images.

Dictionary-based Correction Methods

In many cases, no amount of image correction will improve OCR output accuracy. This is common when scanning unusual fonts and symbols or shapes. Rather than focusing on the input, another transformation layer can be added to the output itself. Assuming the results of an individual scan has a predetermined set of possible values (e.g. only numeric characters), it should be possible to determine which of these values is the most likely interpretation based on some distance metric between the two strings. This requires three components: the OCR output string x, the set of possible distinct string values $Y = \{y_1, y_2, \dots, y_n\}$, and the distance function $f(x, y)$ to compare the two. This should provide a result set Z of similarity measurements such that $z_i = f(x, y_i), 0 \leq z_i \leq 1$. The program then selects the optimal string y_i as the string corresponding to the highest similarity determined as $z_i = argmax_y f(x, y)$. It should be assumed that $f(x, y) = 1 \leftrightarrow x = y$ given all distinct values of y and $|x| = 0 \land |y| > 0 \to f(x, y) = 0$. Lastly, $f(x, y_i) = f(x, y_j) \nrightarrow y_i = y_j$. Note that this requires additional input on behalf of the user which may not be possible in all circumstances.

Implementation and Limits

For this project, the **Levenshtein ratio** is used to calculate the relative distance between strings. The **Levenshtein distance** is calculated as the total number of edits needed to one string to match the other, where an edit may be a character insertion, deletion, or substitution, whereas the Levenshtein ratio divides this distance by the combined lengths of both strings. Other string distance methods may be considered such as **Damerau-Levenshtein distance** (which allows transposition for adjacent characters as those are common in human spelling errors). Given the independence of individual character errors relative to their neighbors in OCR, such methods may be less accurate here.

Testing with the Levenshtein distance method showed promising results when applied to dictionaries with large, varied strings. The list of skill names from *Monster Hunter Generations Ultimate*, along with the possible numeric ranges (-10 to +13 excluding 0, includes sign on positive values) and slot symbol values ("---" for zero slots, "o--" for one, "oo-" for two, "ooo" for three) were used as dictionaries for the image "mhgu2.png" to correct predictions made over 10 thresholds. Without dictionary support, the reported accuracy was 86.41% (89.74% for skill name columns 1 and 3). With dictionary support, accuracy improved to 92.96% overall (100% for skill name columns). All errors stemmed from the software's inability to properly identify symbols such as "+", "-", and "o" in the 2nd, 4th, and 5th columns, as well as the dictionaries' failures in handling distance ties. For example, if the OCR output of a scan is "5", then the dictionary cannot determine whether "-5" or "+5 is more accurate. An examination of each dictionary suggests that accuracy improves when using dictionaries with longer string lengths and a lower mean distance ratio between all possible combinations. For example, a list of all U.S. state names ("Massachusetts", "Louisiana", etc.) would see far better results than a list of their abbreviations ("MA", "LA", etc.). Similarly, a **Levenshtein distance matrix** as a heatmap may be used to visualize the overall similarity between possible terms.

Character-shape Tiebreaking

In cases where the Levenshtein distance is unable to select a single maximum, an alternative tiebreaking solution may be applied at a character level. Rather than comparing the number of edits needed to correct a word, the images of the dissenting characters themselves are compared. For example, "+4" and "-4" both have a Levenshtein distance of 0.5 with "$4". In such a case, the dictionary would normally return the first match by default. Implementing the character-shape tiebreaker would instead try to compare the shape distance between "$" with "+" and "-", where each potential shape is drawn (using either a provided font or some common font with similar shapes) as a numeric matrix. The simplest method for determining distances is to sum the square difference at each pixel, though additional methods and transformations may also be implemented. The character with the smallest distance is then selected for the final predicted result. Note that this method is sensitive to the style of the font itself: variations from serifs, character designs (compare "a" with "**ɑ**" or "0" with "0"), and nonstandard characters (Greek letters, accents, umlauts, etc.) will have a significant impact on the similarity between various characters.

This method has no effect on OCR empty string results, nor can it improve accuracy against missing characters – a result of “4” would still be unable to differentiate between “-4” and “+4”. One solution: assume that a minimum string length must be met based on the shortest item in the dictionary. If the OCR result produces a string shorter than the minimum, assume that the empty characters must be replaced with the character with the most empty space from the set of possible characters in that position. The test image “mhgu2.png” was able to achieve 100% accuracy when utilizing this method to account for the inability to offer corrections with the dictionary method. Notably, this method was shown to improve accuracy when ignoring the character filter option in Tesseract itself, which appears to favor dropping invalid character strings rather than approximating them.

Conclusion

Though the final platform has not fully developed for a truly autonomous text extraction method, the methods utilized in this project demonstrate the viability of such a program. More testing with other sample images and methods is required, however, as is the necessity for algorithms to optimize performance speed and accuracy. Experiments on videos and the development of video-specific methods should be considered. In addition, further structural improvements should be made to the code itself to enable software creation (along with the implementation of a UI). The originally-proposed “spot the differences” feature for comparing similar images and the collection of unstructured lists may be implemented as well with many of the same methods discussed here.