The University of Hong Kong
COMP4801
Final Year Project
Final Report

Athletics live-streaming graphics playout, data collection and information system

Tam Yu Kit
3035555274

Supervisor: Dr. K.P. Chan
Table of Contents

Introduction ................................................................................................................................. 4
Project Background ..................................................................................................................... 4
Project Objective ......................................................................................................................... 4
Potential applications ................................................................................................................ 4
Project background and literature review .................................................................................. 5
Motivation and what is available now ........................................................................................ 5
Literature Review ......................................................................................................................... 5
OCR Technology ......................................................................................................................... 5
Scoreboard Recognition .............................................................................................................. 6
Project methodology .................................................................................................................. 7
System architecture design .......................................................................................................... 7
Central data server design ........................................................................................................... 7
Graphics playout (display) component ........................................................................................ 8
Graphics playout controller ......................................................................................................... 8
Commentator Information System .............................................................................................. 8
Communication between the different components ....................................................................... 8
APIs for integrations ..................................................................................................................... 8
OCR Component ........................................................................................................................ 9
Large scoreboard (Track events) .................................................................................................. 9
Remaining laps indicator ............................................................................................................ 9
Field event scoreboard ................................................................................................................ 10
Non-athletics sports scoreboard .................................................................................................. 13
Summary of scoreboards ............................................................................................................ 14
Identifying 7 segment numbers .................................................................................................. 15
Software Implementation of OCR component ............................................................................. 17
The user flow and the developed OCR component product ...................................................... 27
Experiments and Results .......................................................................................................... 32
First approach of carrying out OCR on 7 segment scoreboard on numbers ................................ 32
Tesseract ...................................................................................................................................... 32
Segment detection with simple algorithm ................................................................................... 32
Obstacle detection ....................................................................................................................... 33
Results testing (manual 7 point OCR system) .......................................................................... 33
Second approach of carrying out OCR on 7 segment scoreboard .............................................. 36
Solving the challenge - accuracy of Tesseract: ........................................................................... 39
Solving the challenge – Deskewing the image............................................................................. 39
Challenge – Generalized obstruction detection ................................................................. 45
Attempt 1 – Feature matching .......................................................................................... 45
Attempt 2 – Color difference .......................................................................................... 46
Solving the challenge – performance of Tesseract API .................................................... 48
Challenge – Fonts of scoreboard ..................................................................................... 53
Performance ................................................................................................................... 54
Final testing result of the end product on scoreboards .................................................... 54
Conclusion and Future Works ......................................................................................... 54
Conclusion ....................................................................................................................... 54
Future work – Retraining tesseract on manual scoreboard fonts ...................................... 55
Future work - Auto-scoreboard discovery ...................................................................... 55
Future work – internally generate more versions of the image for recognition ................ 55
Future work – Improving OCR correctness based on race logic ...................................... 55
Future work – complete the rest of the information system .......................................... 56
Works Cited ..................................................................................................................... 57
**Introduction**

This project aims to automate the athletics live-streaming process through automated data collection running Optical Character Recognition on scoreboards, combined with an information system that can allow display, edit, view of the collected and pre-imported data for both live-streaming and third-party integration via API.

I was responsible for designing the system, and development of the OCR component of this project.

**Project Background**

When an athletics event is being streamed, one of the challenges is to generate and display the information of the race (participating athletes, time, record, athlete performance) in the form of graphic in real time. (E.g. the athlete’s names for the event, the distance of the throw/jump, similar to Olympics live broadcast).

Similar challenge also applies to other sports events including basketball, volleyball, handball, or any sports that involves a scoreboard.

There is currently no integration between the judges who record the results, the judges who run the race, and the live-streaming team who broadcast the event live. The live-streaming team relies on traditional methods including looking at the manual results board that is operated by the judges manually, and manually inputting the data into the graphics system if they would like to show any graphics in the livestream.

Therefore, this project aims to automate the scoreboard results capturing process, and allows the data to be presented in a livestream in the form of computer generated graphics in a convenient, reliable, and flexible manner.

The automatically captured scoreboard results must work reliably, as any incorrectness of the race results will lead to confusion and poor viewing experience for the viewer. Therefore, capturing the scoreboard results reliably is one of the main challenges in this project.

**Project Objective**

This project aims to automate the process of data collection, capturing, and displaying through computer vision technology. The data is then used to generate graphics for live streaming. Reducing the amount of manpower needed in order to run a livestream. And use computer vision technologies to assist the livestream team to reduce human errors, while adding even more graphics and information onto the livestream video. Allowing the viewers to enjoy an athletics or other sports livestream with more information, while the livestreaming team can operate with less manpower.

**Potential applications**

Potential applications of this project include live streaming for a variety of sports competitions that involves a scoreboard, including athletics track and field, basketball, handball, volleyball, hockey, and virtually any sports that shows results in the form of a scoreboard. Compared to currently available options on the market, it can make live streaming with OCR much more accessible.

Moreover, since the live score results can now be obtained automatically, it can be used to develop other products. For example, a website that allows users to view the match results in real time, as the race progresses. This allows the real-time match results to be transferred over narrowband networks, as only text information has to be transferred.
Sports journalists can also make use of the time tagged scoreboard information to quickly find the photos or sections of video that they need. Since they now know the time of a certain match result. An example would be finding the time when the match score equalized, and identifying the athlete who scored the equalizing score through a recorded video of the match easily. Without having to review the whole video to find out the time when the match score equalized.

Project background and literature review

Motivation and what is available now
I chose to do this project as there is virtually no freely available software in the market that is designed for scoreboard Optical Character Recognition (OCR). Moreover, the commercial software that is available on the market is highly priced. Making this technology hard to access by live-streaming teams with lower budgets.

An example would be “scoreboard-ocr.com”. Their software has a yearly subscription cost of 390USD per year.

An open-source alternative would be “Scoreboard-webcam-OCR” by “xyk2” on GitHub. However, different from this project, it does not have integration with athlete or race information. It also only supports electronic LED scoreboard. Traditional manual scoreboards are not supported. This limits the use case of “Scoreboard-webcam-OCR” greatly.

Literature Review
Various papers have explored the use and performance of different OCR technologies in different applications, and some specifically on sports. They will be discussed in detail below.

OCR Technology
A study by Majumdar and Gupta in 2019 explored the accuracy of various OCR tools on different input images. The study studied the accuracy of Tesseract, Gocr, Ocrad, and TensorFlow, and found that the accuracy of various OCR tools varies depending on a number of factors. Including type of input image (clear black on white text, or color image of a vehicle number plate), whether background is cropped, and whether shadows exist in the image. (Majumdar & Gupta, 2019)

The research found that when a clear image of black text on white background was passed to the OCR tool, the accuracy of the recognized text by different OCR tools is consistently much higher than a color image of vehicle number plate. Tesseract performed the best in a test with 190 images of black and white text, with an accuracy of 96.31%. Followed by Gocr with an accuracy of 92.63%, TensorFlow with an accuracy of 88.42%, and Ocrad with an accuracy of 53.15%. (Majumdar & Gupta, 2019)

The study also carried out a test of with 11 color vehicle number plate images, with the background cropped. The results showed that Tesseract is the best performing OCR tool, with an accuracy of 90.95%, followed by TensorFlow with an accuracy of 72.72%, and Gocr and Ocrad at 54.54%. (Majumdar & Gupta, 2019)

The study concludes that Tesseract is the best performing OCR tool. However, the study also mentioned that the input image must first be pre-processed by deskewing, binarization through thresholding, and segmentation before passing to Tesseract. Otherwise, it will lead to very poor output. (Majumdar & Gupta, 2019)
Another study by Agbemenu, A. S., Yankey, J. & Addo, E. O studied the use of Tesseract to recognize car number plates with the use of OpenCV. (Agbemenu, Yankey, & Addo, 2018) The goal of my project to recognize scoreboards on the sports ground is similar to recognizing car plates in an image to a certain extent. Both require locating a target object (e.g. car number plate) in a given image, and carrying out OCR on the object.

The researchers proposed a system that first carries out candidate detection on the input image that is greyscaled and smoothened with Gaussian blur. Candidate detection is carried out by edge detection and template matching using trained classifiers. Character segmentation is then carried out by cutting out each character, and passing each of the characters to the Tesseract engine that was trained with car number plate fonts and a dictionary of possible car number plates. (Agbemenu, Yankey, & Addo, 2018)

The results showed that among 500 test images, candidate detection using edge detection had a success rate of 79.4%, while feature detection was slower but had a detection rate of 90.8%. OCR successfully recognized approximately 60% of the plates. The study also carried out a failure case analysis and found that faded characters, decorations on the car plate, and noise due to dirt, affected the segmentation process significantly, and in turn affected the OCR results. (Agbemenu, Yankey, & Addo, 2018)

This study reveals that it is possible to develop a similar program using Tesseract and OpenCV for identifying scoreboards. Moreover, character segmentation, retraining Tesseract on the target font, can also improve the performance of OCR. It also sets an approximate expectation that recognition success rate of 60% is achievable. Most critically, it suggested that to achieve a high OCR success rate, the input image should not have faded characters or additional decorations, otherwise the plate segmentation process will be affected negatively.

However, since my goal is to achieve a much higher scoreboard OCR success rate of at least 95%, combined with the nature that the scoreboard and camera do not move once it is set up in real world applications. I decided to let the user specify the 4 corners of the scoreboard to achieve the best possible image for OCR, in order to achieve a high OCR success rate.

Scoreboard Recognition

A study by Hsieh, Huang, and Hung studied the use of OCR on scoreboards in broadcast baseball videos. The goal of the study was to carry out scoreboard OCR on baseball broadcast program on different broadcast channels including ESPN, FOX, YES. (Hsieh, Huang, & Hung, 2008)

The researcher proposed a solution that first identifies the scoreboard type using template matching, which provides the position information of the text. Digit recognition is then carried out by first pre-processing and then recognition through feature extraction. The pre-processing step includes scaling up the image, binarization, erosion to make the text “thicker”, size normalization so the image size of the digit is normalized. Digit recognition is carried out through extraction of contours through a neural network classifier. (Hsieh, Huang, & Hung, 2008)

The research also proposed to increase the accuracy of the OCR result through “Multi-frame majority decision”. Since the scoreboard data is generally kept the same for a long time, majority voting over several consecutive frames can be carried out, to prevent individual errors in a particular frame from being accepted as a valid OCR result. (Hsieh, Huang, & Hung, 2008)

Results showed that the proposed solution achieved an accuracy rate of 99% in recognizing the scores of two teams in 1280 samples, compared to 85% with commercially available OCR SDK. It also
recognized the number of outs with 98% accuracy in a test of 640 samples. Compared to 83% accuracy using a commercially available OCR SDK. (Hsieh, Huang, & Hung, 2008)

Although this article explored OCR on broadcast scoreboards, it shows that it is possible to achieve a very high accuracy of scoreboard OCR, using a combined approach of recognizing numbers, and filtering incorrect results through majority voting.

From the literature reviews above, it can be concluded that Tesseract is the best performing yet non-commercial OCR engine. Therefore, Tesseract was chosen as the OCR engine for this project. Moreover, pre-processing must be carried out for Tesseract to recognize the text properly. Including cropping the background, binarization, cleaning up of any distractions or noise.

Literature review also suggests a benchmark to meet for the accuracy of this project. When training using the same font is possible, combined with a “multi-frame majority decision” approach, a 98% accuracy is achievable. Studies on vehicle license plate recognition suggest that OCR accuracy of approximately 60% is possible in real life.

Project methodology
System architecture design
The system architecture design is completed as follows. The system is divided into 5 major components in a microservice-like design. This is to minimize the dependencies between the different components.

Central data server design
The central server was implemented using PHP and MySQL. MySQL was chosen as it is a relational database system. Making it very convenient for updating, accessing, and modifying data that are linked to each other. An example would be: one athlete can participate in multiple races. A relational
database makes modifying the name of the athlete extremely convenient. Whereas, a non-relational database / nosql database makes such update more complicated, as more logic has to be done on the application side. While MySQL takes care of this on the database side.

PHP was chosen as it can be easily integrated with MySQL, and I am very familiar with development in PHP.

**Graphics playout (display) component**

Graphics playout – This acts as a Display Component. It takes the data from the central data server to be displayed in the form of a webpage with custom graphics design. We will make use of web technologies (HTML, CSS, JS) to complete this component.

**Graphics playout controller**

Graphics playout controller— This component aids the user to display their desired information on “Graphics playout component” using prompts and a control interface via a controlling webpage (web control). The live-stream operator can easily show or hide the participants information graphics and details of the race for composing better viewing experience when used as an overlay to the camera video feed.

**Commentator Information System**

The commentator information system is a webpage that can be opened on laptops/tablets. The webpage shows the current event, current athletes’ names, or commentator added notes for a particular athlete. This component is especially designed for commentators to provide more information about the background of the athletes, and the events that are happening in real time.

For example, through the OCR results of scoreboard in field events, the commentator can immediately mention the results of the past trials of the particular athlete, without the commentator writing down the past results manually.

**Communication between the different components**

The components are connected via HTTP or websocket technologies. The idea of “event broadcasting” is used.

When an event is received or triggered by any of the components of the system. E.g. “new race identified” event fired by the OCR component that is identifies the current event when a race is about to begin, the information will be sent to the central data server. The central data server will then broadcast the message “new race identified” to all of the other components. The components will then carry out the relevant activity, or no activity at all, depending on their own function and role.

This design minimizes the dependencies between the components and avoids a lot of cross function calls between each component of the system. This also helps in speeding up the development and avoiding changes in one component breaking another component.

**APIs for integrations**

This component provides integration for other information systems to extract information for a certain athlete, certain race/event, current race, via APIs. It connects to the central data server, and allows the data to be obtained, or updated.
OCR Component

The OCR component is designed to carry out OCR on multiple targets on the sports ground for an athletics event. Including the “remaining laps indicator”, “large scoreboard (Track)”, and “field event scoreboards”. Each of the targets will be discussed in detail below.

Large scoreboard (Track events)

![Example of scoreboard used for track events](image1.jpg)

Figure 1 Example of scoreboard used for track events

Figure 1 shows an example of the large scoreboard that is commonly used in track events in Hong Kong sports grounds. The large scoreboard is an LED TV wall that shows the track event that is in progress. The event name, event record, the name of athlete in each lane, the team, and elapsed time is displayed on the scoreboard.

By running OCR on this scoreboard, the current event name, lane number, event record, elapsed time, and lane athlete name and team information can be obtained, which can then be integrated with other parts of the system for more features.

Remaining laps indicator

![Image showing the remaining laps indicator in the sports ground near the finishing line](image2.jpg)

Figure 2 Image showing the remaining laps indicator in the sports ground near the finishing line
The “remaining laps indicator” is a 7-segment number display that is placed near the finishing line of the sports ground. The display shows the remaining number of laps of the leading athlete in races where athletes need to run more than 1 lap.

For example, if the race is 800 meters, the “remaining laps indicator” will first show the number “1”. Once the leading athlete passes the finish line, the “remaining laps indicator” will show the number “0”. Indicating that this is the final lap that the leading athlete (fastest athlete) needs to run.

For 5000-meter races on a sports ground with 400 meters per lap, the remaining laps indicator will start with the number “12” and decrease by one every time the leading athlete crosses the finishing line, until the number goes to “0”.

Figure 3 Close up of the remaining laps indicator

Field event scoreboard

Figure 4 Field event scoreboard

Figure 4 shows a scoreboard that is commonly used in field events, including but not limited to high jump, long jump and javelin throw. The rectangular scoreboard is mounted on a pole that is on a stand. The judges or operator of the scoreboard will change the numbers manually, and the rectangular scoreboard can be rotated to show the results to people in different directions.
There are two rows of numbers on the scoreboard, which is explained in Figure 5 below.

![Scoreboard Image]

<table>
<thead>
<tr>
<th>Trial No. 1/2/3...</th>
<th>Bib Number</th>
<th>Bib Number</th>
<th>Bib Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>Height (Meter)</td>
<td>Height (Meter)</td>
<td>Height (Meter)</td>
</tr>
</tbody>
</table>

*Figure 5 Diagram explaining the field event scoreboard*

The digit in the first row, first column shows the trial number. In field events, athletes often have multiple attempts to play in the event, and the best performing attempt is counted. Therefore, this number should be recognized.

The digits in the first row, columns two to four are bib numbers. Bib number identifies the individual athlete through a 3-digit number. The numbers “384” is showed in Figure 4, which means that the athlete with bib number 384 is competing.

The numbers on the second row are all digits for displaying the height, or distance, of the current attempt. The numbers “1.78” is showed in Figure 4. This means that the athlete is competing for the height of 1.78 meters.

Therefore, combining all of the information on the scoreboard, we are able to obtain the detailed result of the athletes in field events.

In my preliminary testing, by pre-processing the image by greyscaling and increasing the contrast, the output of Tesesract was not ideal as demonstrated in Figure 6 below. The Tesseract recognized texts were wrong. The failure of Tesseract to recognize the text properly is caused by multiple factors.

Firstly, vertical green borders that separate the individual numbers were recognized as “1” by Tesseract. Therefore, leading to incorrect tesseract output.

Secondly, the horizontal green borders also caused a problem for the recognition of the scoreboard. The green borders were quite close to the numbers. Research shows that Tesseract expects some whitespace as border around the text. Otherwise, Tesseract cannot recognize the text well.
Thirdly, insufficient contrast in the number “8” in the first row, and in numbers “1”, “7”, “8” in the second row caused by aging and yellowing of the plastic on the scoreboard also posed a challenge for Tesseract to recognize the processed image.

Fourthly, low input resolution leads to a lot more noise when it is resized to a larger image. This poses a challenge to Tesseract to recognize the text, coherent with the findings by Agbemenu, Yankey, & Addo, 2018.

To verify the hypothesis above, I cleaned up the image manually, and inputted it to Tesseract again. Figure 7 below shows the output of Tesseract, OpenCV is used to overlay the output to the tesseract positions. Tesseract was able to identify the numbers correctly after manually cleaning up the image. Therefore, this exploration suggests that Tesseract expects a clean image, with no extra elements such as grids, in order to carry out OCR successfully.
Non-athletics sports scoreboard

Figure 8 Example of an electronic 7-segment scoreboard used in a local handball match

Figure 9 Example of a yellow on black manual flip-page style scoreboard used in a local handball match

For non-athletics sports, the same OCR component software can also be used to identify the score of different teams. An example would be in a volleyball or handball match, where a 7-segment digit electronic scoreboard (Figure 8) or manual regular flip-page numerical (non-7-segment, Figure 9) scoreboard is used.

Electronic scoreboards often display more information than the manual flip-page style scoreboard. The scoreboard in Figure 8 shows two rows of text. The first row shows the remaining time of the match, in mm:ss format. Where m represents minutes, and s represents seconds. The image in Figure 8 shows the remaining time of 6 minutes and 28 seconds.

The second row has two sets of numbers, which are “15” and “11”. It shows the current score of the two teams respectively.

Manual flip-page numerical scoreboard (Figure 9) often displays 2 sets of numbers. Each set of numbers represents the current score of the corresponding team, usually from 0 to 30.
Summary of scoreboards
Due to the large variety of scoreboards that are used in different sports, it is not practical to design an independent OCR component program for each specific type of scoreboard. Therefore, I designed a general OCR component that can handle most of the scoreboards discussed above.

The scoreboards can be differentiated by 3 main factors:

1. **Type of font**
   Whether the scoreboard shows 7-segment numbers or numbers and text with regular type font.

2. **Color**
   Whether the scoreboard has “white or light colored text on a black background”, or “black or dark colored text on a white background”.

3. **Alphanumeric or numerical text displayed**
   Whether the scoreboard only displays numerical results, or shows alphanumerical texts e.g. the scoreboard that is used for track events illustrated in Figure 1.

The findings from literature review and my testing above can be concluded as follows:

1. Scoreboard must be first be deskewed.
2. The location of numbers must be specified manually by the user to increase OCR success rate. Unless the localization of the numbers in the image can be identified automatically.
3. No extra elements or noise should be present in the image. Otherwise, Tesseract will perform very poorly.
4. There must be a white border between the edge of the text and the edge of the image.

Deskewing refers to the process of distorting the image of the scoreboard that was taken at an angle, to straighten and recover the original shape of the scoreboard. The scoreboard can be deskewed using OpenCV.

Due to limited time and manpower on development of the OCR component, I decided to let the users specify the location of the numbers.

Extra elements can be reduced by limiting the region that is passed to Tesseract for identification. Through the user specifying the region of interest (ROI) when specifying the location of the text to be identified, the amount of noise and extra elements can be reduced. Moreover, through a combination of gaussian blur and binarization, the amount of noise can also be reduced significantly.

Identification of 7-segment numbers will be discussed in the following section, while testing results of the two 7-segment numbers recognition approaches will be discussed in Experiments and Results below.
Identifying 7 segment numbers

Carrying out OCR on 7-segment numbers is different from regular printed numbers, as it is of a very different font compared to regular font.

7 segment numbers are composed of 7 “sections”, or “segments”. The on/off state of each of the segment determines the number. An example can be seen in Figure 10 below. Figure 10 shows a 7-segment display that is displaying the numbers “4 3 2 1”. The dark grey rectangles / segments are “off”, and the yellow rectangles/segments are “on”.

![Example of simulated 7 segment display, showing the numbers “4 3 2 1” from left to right](image)

I tested Tesseract with the regular language pack, with images of 7-segment numbers, and it showed that Tesseract was unable to recognize the 7-segment numbers properly.

Figure 11 shows a digital scoreboard with 7-segment numbers. The rectangles and numbers in blue represent the regions of interest (ROI) that the user has marked. The image is then cropped to the region of interest and passed to Tesseract for OCR recognition after rescaling to a height of 36 pixels. Tesseract was using the regular default language pack, a whitelist of numerical characters was set to help Tesseract recognize the correct numbers, and the Page Segmentation Mode (PSM) is specified to be “single line” to help Tesseract recognize that there is one line of text in each ROI. The text in green shows the output of Tesseract.

It is obvious that Tesseract was unable to recognize the 7-segment numbers correctly. For example, the number “6” in the first row of the scoreboard was not identified. While the number “07” in the first row of the scoreboard was recognized as “9” with a confidence of 87%. The same problem applies to the second row of numbers, the number “15” was recognized as “5” with 30% confidence, and “12” recognized as “2” with 0% confidence.

Figure 12 shows the same scoreboard with the numbers “6 25” in the first row, and “15 11” in the second row. Tesseract was unable to recognize any of the numbers in Figure 12.
Since I have confirmed that Tesseract cannot recognize the 7-segment numbers correctly from the discussion above, it is clear that another approach had to be taken.

I proposed two approaches to identifying 7-segment numbers. The first approach is to identify the numbers by the user locating the 7 segments of the number manually, and through detecting the on-off state of each of the segments, the number can be identified.

The second approach is to use Tesseract with a 7-segment language pack to carry out the identification process. It will be slower when compared to the first approach described above, as Tesseract uses a neural network to identify the numbers. However, it should be more robust when compared to the first approach.
Software Implementation of OCR component

I developed the OCR component. The OCR component is written in **Python**, using the library **OpenCV** for video/image input, processing, sliders GUI, **Tkinter** for the start screen GUI, and **Tesseract** is the OCR (Optical Character Recognition) engine used for this project.

Python was chosen as it speeds up the development speed significantly, compared to C++. Moreover, most of the resources on OCR and Tesseract are based on Python. Making it the optimal programming language to be used for this project. Especially when I do not have any experience with OCR, nor OpenCV in the past.

**Tkinter** was used to develop the GUI for the start screen, as it is cross platform, and is natively supported by Python. Yet provides sufficient features to create a GUI for the OCR component of this project.

The downside of Python is it does not run as fast as C++. However, running speed is not an important concern for this project. Moreover, the performance difference between C++ and Python is not significant enough to be a concern for this project.

**Tesseract** is an OCR engine that was initially developed by Hewlett Packard in 1980s and was open sourced in 2005. Tesseract runs as an executable that must be installed on the same PC as the Python program. Moreover, it allows the user to provide additional language packs to recognize text that is not initially supported, for example, 7-segment characters. It was written in C/C++. Therefore, two python libraries **pytesseract** and **tesserocr** were used to integrate the Python code that I wrote with Tesseract. The python libraries are an API that makes advanced integration with Tesseract possible.

*The processing behind (Image Pre-processing)*

Combining the information from literature review, tutorials on the internet, code on GitHub, documentation on Tesseract and my own exploration and testing, I found out that the image must be pre-processed first, before passing to Tesseract for recognition. A study by Majumdar & Gupta, 2019 stated that “If the input images are not pre-processed Tesseract tends to give very poor output”, and my test results were coherent with this finding.

The following image pre-processing is carried out in my OCR component system. They were compiled from my testing and research on sample implementations of OCR with OpenCV and Tesseract.

**Step 1 – Deskew.**

Deskew is the process of distorting the input image to a “straightened” angle. It is necessary as Tesseract is unable to recognize text in images where the text is at an angle.

An example is illustrated below.

Figure 13 shows the original image from the camera of a scoreboard that was placed at an angle to the camera. Figure 14 shows the OCR results of the input image in Figure 13 after a series of pre-processing including binarization and rescaling. The blue rectangles in Figure 14 are the regions of interest that is specified by the user. Results show that Tesseract is unable to recognize the numbers “1” at all. Number “2” was incorrectly recognized as number “1” with a confidence of 77%, and number “5” was recognized correctly with a confidence of 88%.
Compared to a deskewed version of the image illustrated in Figure 15, the OCR results were completely correct. This shows that it is necessary to deskew the image in order for Tesseract to carry out OCR successfully.

Figure 13 Skewed scoreboard input example

Figure 14 OCR Result of skewed scoreboard input example

Figure 15 OCR results of the deskewed scoreboard sample
How Desking is carried out:

Deskewing is carried out by using two pieces of information:

1. The location of the 4 corners of the skewed scoreboard
2. The output location (destination points) of the 4 corners of the deskewed scoreboard (the 4 destination points form a proper regular rectangle)

Point 2 above implies that the original aspect ratio of the scoreboard must be known in advance for an accurate deskew. However, since we do not know the original aspect ratio of the scoreboard, must be guessed.

I used the function `cv2.minAreaRect()` to find the minimum bounding rectangle around the user specified 4 corners of the scoreboard. From the returned minimum bounding rectangle, we are able to make a best guess on the original aspect ratio of the scoreboard, and use this information to determine the output size (resolution) of deskew.

Using the function `cv2.getPerspectiveTransform()`, a transformation matrix is obtained. The matrix “stretches” the 4 original corners to the regular rectangle’s corners to achieve deskew.

Then, the function `cv2.warpPerspective()` is used to apply the transformation to the image with the transformation matrix obtained from `cv2.getPerspectiveTransform()`, and a deskewed image is obtained.

Demonstration of the following steps 2 onwards will use the test image below:

Step 2&3 – Greyscaling and Binarization

Greyscaling is the process of converting a color image comprised of red, green, and blue channel information into greyscale.

Greyscaling was carried out with OpenCV’s function `cv2.cvtColor(deskewedImage, cv2.COLOR_RGB2GRAY)`. This function takes an inputImage and returns an image that is converted to grey. This allows the next step, binarization (or thresholding) to be carried out properly.
After greyscaling, the image looks as follows. Tesseract was unable to recognize any text, given the two ROIs:

![Image of two gray boxes with the number 1 and 2]

Binarization (or Thresholding) an image, is the process of binarizing an image pixel that initially has 255 levels, in the value of 0 to 255, to bi-level image, in the value of 0 or 255. I implemented binarization using the OpenCV function

```c
cv2.threshold(inputImage, thresholdingValue, maxValue, type)
```

This function takes an input image, and returns an image that is binarized according to the thresholdingValue specified. The “type” value determines whether to set the output image to the maxValue, or 0, if the input pixel value is higher than the threshold.

To accommodate both scoreboards with white text on black background and or black text on white background, the user has to choose the operation mode of the OCR component first at the start screen of the OCR component GUI. The argument that is passed to the “type” field in the function `cv2.threshold()` is `cv2.THRES_BINARY_INV` if the operation mode of “white text on black background” is chosen, and `cv2.THRES_BINARY` is the operation mode of “black text on white background” is chosen. This is because Tesseract expects black text on a white background to function properly. Therefore, the thresholding must be inverted if the input text is white on black.

Since the thresholding value used depends on the lighting conditions of the video of the scoreboard, I have designed a slider in the UI that allows the user to change the threshold in real time during OCR.

This adaptive design allows the same OCR component to be used across different lighting situations and different scoreboards.

After thresholding, the output image looks as follows. We can see that Tesseract is still unable to recognize the text in this example. Therefore, more processing needs to be done.

![Figure 16 Thresholded example image](image)
Step 4&5 – Erode and dilate:

Erode and dilate is the process of increasing or decreasing the white areas, and effectively changing the thickness of the text. This allows text that is too thick / too thin to be recognized properly.

In the case of black text on white background, erosion increases the thickness of the black text as demonstrated below.

After erosion of Figure 16, the image will look as follows:

![Image of eroded text](image)

As for dilation, dilation makes the thickness of the black text appear thinner. Therefore, making it easier for Tesseract to recognize if the text is too thick.

After dilation on Figure 16, the image looks as follows. The text becomes a lot thinner compared to Figure 16:

![Image of dilated text](image)

Step 6 – Crop to region of interest and resize

After processing the steps above, the program will crop the image to region of interest, and pass it to Tesseract for recognition.

Cropping the image to region of interest is necessary as Tesseract did not perform well in my testing with Tesseract without cropping the image. Tesseract often seems to have difficulty finding the location of the text for OCR, and therefore may return text that is incorrect, or does not return any text at all. This finding is consistent with the study carried out by Majumdar & Gupta in 2019, they found that “the background had to be cropped as the accuracy went very low” when using Tesseract.

After cropping the image to the region of interest, the image is resized to a height of 36 pixels. This resize is necessary as counterintuitively, Tesseract did not perform well with large images with a height of more than 100 pixels. This is consistent with the forum discussion on Tesseract by Willus...
Dotkom. The forum user carried out an analysis on Tesseract’s accuracy on different text size and resolution. The results show that the error rate is close zero when the text height is 30-33 pixels. (Dotkom, 2018) Users may draw select the region of interest boundaries fairly larger than the size of the numbers, instead of drawing a tight bounding box, as it will allow some room for the camera or the scoreboard to move without the need to redraw the regions of interest. Therefore, a higher value of a height of 36 pixels was chosen.

After resizing, Tesseract is now able to recognize the numbers “1” and “2” with a confidence of 64% and 67% as follows. This is not ideal, as it would be hard for the program to tell whether the output is reliable.

![Image of cropped and resized image](image)

The cropped and resized image that is passed to Tesseract for recognition is as follows. We can observe that there are some pixels that are in grey (pointed by the red arrow), but should be black. Therefore, Thresholding is carried out again, as Tesseract expects a binarized image. An image is not binarized if there are grey pixels in it.

![Example image that is cropped to ROI, resized](image)

*Figure 17 Example image that is cropped to ROI, resized*
Step 7 – Thresholding after resizing
Thresholding is necessary after resizing the image, as the image may be “blurred” after resizing due to resampling of the image when resizing. Therefore, thresholding is carried out on the resized image to further clean up the image before passing it to Tesseract for recognition. This allows Tesseract to take in a binarized image and improves the accuracy of Tesseract’s output.

The thresholded image looks as follows. Compared to the image in Figure 17, we can observe that the number “2” now is filled in solid black, ideal for Tesseract to recognize on. There are no more grey pixels.

![Thresholded Image](image-url)

Step 8 – Pad border
After cropping and resizing, border padding is carried out. A white border with a width of 2 pixels is padded in default, to all 4 sides of the image that has a height of 36 pixels.

Border padding was carried out to increase the success rate of Tesseract recognizing the numbers correctly, as Tesseract does not perform well on input images with no whitespaces in my testing. Which is consistent with Tesseract’s documentation. Tesseract expects a small white border around the text.

A border was padded to the image in case the user draws a tight region of interest around the text, which will cause the text to be too close to the edges, and leading to poor Tesseract performance.

Padding borders can also allow Tesseract to continue to be able to recognize the text on the scoreboard, even when the text is too close to the edge of the image due to vibrations of the camera.

The border width can be adjusted on-the-fly by the user via a slider in the GUI during OCR.

The necessity of border padding is demonstrated below. When the user draws a very tight ROI in Figure 18 below, the number “2” is recognized with a confidence of only 39%. This is due to lack of borders in the resized and processed image that is passed to Tesseract (Figure 19). The low confidence makes it difficult for the program to decide whether the OCR result should be accepted. Therefore, I try to increase the Tesseract’s recognition result confidence by optimizing the image.
Figure 18 Tight ROI example

When we pad white boarders of width 2, the confidence is increased to 76% as shown in Figure 20.

Figure 19 Image passed to Tesseract, tight ROI

Figure 20 Borders padded and image passed to Tesseract for recognition
When the width of border padded is increased to 8 pixels, the confidence is increased to 97% as shown in Figure 21.

![OCR in Progress](image1)

![OCR Debug](image2)

Figure 21 Tight ROI OCR result with border width of 8 padded

Therefore, border padding is useful as shown above, to allow Tesseract to recognize the text with high confidence when the ROI is drawn tightly.

The steps above summarize the pre-processing that is carried out on the image before it is passed to Tesseract. The steps are essential for improving the accuracy of the output of Tesseract, making the OCR component produce results that are accurate enough for use in a sports livestream. Next, OCR results processing is carried out, which will be described below.

**OCR Results processing and sending to server**

The Python library `tesserocr` was used for integrating Python and Tesseract. `tesserocr` is used to access the Tesseract Base API in order to interact with Tesseract.

After Tesseract attempts to recognize the text in the image, a result consisting of 3 pieces of information is returned.

1. Recognized text
2. Results confidence
3. Bounding box of the location of the recognized text

Recognized text is obtained by Tesseract Base API’s `api.GetUTF8Text()`. It returns a UTF8 text string of the text that Tesseract recognized in the given image. And it returns a string of length 0 if no text is recognized by tesseract.

Results confidence is obtained by Tesseract Base API’s `api.MeanTextConf()`. It returns the (average) confidence value of the output text, the value ranges between 0 and 100.

Bounding box is obtained by Tesseract Base API’s `api.GetComponentImage()`. It gives boundaries of each level kind component, e.g. textline. This allows the in depth debugging and analysis of Tesseract’s output, as we can see where in the image, did the returned text come from.

When the scoreboard is in the process of updating, for example a human flip the scoreboard pages in a manually operated flip-page scoreboard, or the number is being updated in a 7-segment scoreboard, incorrect OCR results may occur as a result of the distraction. Figure 22 below shows a hand flipping the page on the left on a manual flip-page style scoreboard. The previous value is ‘1’,
and the distraction action of hand flipping the page leads to a value of ‘4’ to be recognized with 95% confidence. Figure 23 shows the same scoreboard without distraction for comparison.

![Figure 22 Example of distraction - “hand flipping the pages” leading to incorrect OCR results](image)

![Figure 23 Scoreboard with distraction for comparison](image)

Therefore, to minimize the effect of such disturbances on the result OCR results, a combined approach was taken.

The confidence of the OCR result, and the past 7 successfully recognized text were considered. OCR result is separated into “current frame OCR result”, in which only the OCR result of the current frame is considered; and “stabilized OCR result”, which is formed by the past 7 accepted “current OCR results”.

When
1. “Current frame OCR result” is not blank, and
2. Tesseract output confidence is higher than or equal to the acceptance confidence threshold, the current OCR result is considered as accepted. Accepted “Current frame OCR result” are populated to a python list of length 7.

The python list mentioned above stores the past 7 “current frame OCR result” that is accepted. When all of the elements in the python list are identical, a “stabilized OCR result” is produced.

Through this combined approach, it prevents temporary disturbances from affecting the “stabilized OCR result”. Therefore, the accuracy and correctness of the OCR results is increased.
This approach works in filtering out incorrect OCR results caused by disturbances such as temporary obstructions, or update process of the scoreboard. As the OCR result when disturbances are occurring usually has a significantly lower confidence level (90% for non-obstructed, versus 70% for obstructed/disturbed OCR output). Moreover, the OCR output text also changes from frame to frame, when disturbances are happening. Therefore, such combined approach considering both confidence and past OCR results (stabilized) works to filter incorrect OCR results caused by disturbances.

When a change in the “stabilized OCR result” is detected, the new value will be sent to the server. This “send on change” approach reduces the amount of communication between the server and the OCR component significantly. If the OCR result is sent to the server every time an OCR result is in a stabilized state, it will cause a lot of requests per second to the server.

The user flow and the developed OCR component product
Following the pre-processing flow above, and results processing, the user flow can be concluded as follows.

The completed OCR component works as follows to the user:

1. User makes configuration on camera ID, resolution, server URL, and select the operation mode (scoreboard target type: black on white/white on black, 7-segment / regular text, alphanumeric or numerical only) on the launcher window.
2. The user can optionally preview the camera to help to choose the camera ID. Camera ID is depending on the system. It can be previewed by pressing “PREVIEW Camera with current settings” button at the bottom left of the launcher window. After confirming the camera, the user can press ESC on the keyboard, or simply close the window to exit camera preview mode.

3. User confirms the configurations and starts the OCR

![Scoreboard OCR system launcher](image)

4. A help message window appears, providing instructions on how to specify the scoreboard corners.

![Pick scoreboard corners](image)
5. User specifies the 4 corners of the scoreboard for deskewing. Help text is shown on the top left corner of the image.
(corners denoted by green dot)

6. User specifies the 4 corners, and the program will automatically proceed to the next step once all 4 corners are specified.

7. A help window appears, notifying the user of how to properly deskew the scoreboard, if automatic deskew failed to automatically deskew the scoreboard properly.

8. The result of automatic deskew is displayed. The user can manually adjust the width of the image, using the slider in the window titled “Adjust aspect ratio” if automatic deskew result is incorrect.
   A help text also tells the user to press ESC key on keyboard when adjustments is completed.
9. User specifies the regions of interest (ROI). i.e. the location of the digits in the deskewed image. It is not necessary for each ROI to contain 1 number. Recognition of multiple numbers, or characters, is possible. As long as there are no obstructions, e.g. a black border between numbers on a white background. However, text that is recognized from the same ROI will be sent to the server as one OCR result.

The user uses the number keys 0 to 9 on the keyboard to choose the ROI number first.

When the user presses the key ‘0’ on the keyboard, the user can specify the region of ROI0, and vice versa.

The user can then draw a rectangle by clicking and dragging the mouse on the image. A red rectangle with the ROI number appears on the screen, denoting the ROI that the rectangle belongs to.

If the user makes a mistake, the user can simply redraw the rectangle by clicking and dragging again, or press “C” on keyboard to clear the ROI specified.
10. After specifying all of the ROIs needed on the UI, the user can press “Q” on keyboard to exit ROI specification mode and proceed to OCR mode.

11. System constantly runs OCR on the user specified ROI on the deskewed image. 3 windows are displayed. The window “OCR in Progress” shows the current live video feed of the camera selected, the specified ROI(s) is shown in blue, and the OCR results that are higher than the confidence threshold set by the user is displayed in green, with the value in the first row of the green text, and the confidence (in %) displayed in smaller text below the OCR value. OCR results that are lower than the user specified confidence threshold is displayed in red. Users can also change the binarization threshold value, erosion, dilation, border padding, and confidence threshold in real time by adjusting the sliders in the window titled “OCR Live Config” The OCR component operation status is also displayed in the “OCR Status Monitor”.

12. The system also sends the new OCR results to the server if a new OCR result is found for any ROI.
Experiments and Results

First approach of carrying out OCR on 7 segment scoreboard on numbers

Tesseract

I first tried to preprocess this image to a binarized image, and pass it to Tesseract for recognition, using the default language pack. However, Tesseract was unable to recognize the numbers at all. Therefore, another approach must be taken.

Segment detection with simple algorithm

My first approach to 7 segment display number is a rather simple method. Taking advantage of the 7 segments are at a fixed position in ideal conditions, identifying the numbers should be as simple as detecting on the on/off state of each of the segments. Starting with a video input image (Figure 26), the user first clicks on the 7 segments in a specified order for each number (denoted by green dots in ). Then, the program will binarize the image through thresholding, by thresholding a value that the user has set using a slider in the program.

A very basic obstruction detection is also done. Obstruction detection detects whether the camera’s view of the scoreboard is obstructed partially or completely. It prevents incorrect OCR results caused by obstructions to be recognized and accepted as “correct”. The image on the left in Figure 24 shows the number “4” displayed on a 7-segment display. However, if the left section of the number “4” is obstructed by a dark colored obstacle denoted by the dark green oval on the right of Figure 24, the number will appear to be a “1” to such algorithm.

Therefore, some reliable obstacle detection must be in place.

![Figure 24 Obstacles lead to incorrect OCR results](image)
Obstacle detection

My first attempt in obstacle detection was through detecting the brightness of two user specified points on the image, denoted by the two green dots in the red circle in Figure 25. The user specifies two locations in the input image that is always black, regardless of what number is displayed, after thresholding and binarization. With this assumption, if the thresholded brightness value of the two pixels is not black (0, 0, 0), and is instead white (255, 255, 255), then there must be an obstruction. When there is an obstruction, the OCR results may be ignored.

![Figure 25 First attempt in 7 segment number recognition](image)

Results testing (manual 7 point OCR system)

To test the user experience and performance of the 7-segment OCR system in real life, I went to a local handball competition that uses a 7-segment scoreboard to capture some video recordings to test the performance of the program in real world.

**Equipment used:**

Video camcorder – Sony PXW-Z90 recording at 1080p, H.264 (1920 pixels wide, 1080 pixels in height), at 50 frames per second, with 15x zoom.

Image stabilizer enabled. (Sony Steadyshot Standard)

Tripod: Manfrotto BeFree carbon GT.

The camcorder was placed on the tripod, directly across the scoreboard, and not moved throughout the competition.

Focus and exposure: The camera was set to manual focus, and manual exposure.
This test provided very valuable insights on the limitations of such system design and implementation that would be a problem in real world applications:

1. Video shakiness due to floor shakiness, leading to incorrect OCR results
2. Obstruction leading to incorrect OCR results
3. “countdown timer” does not always count down. It may be increased manually by the race judges in special situations.

We will now discuss the limitations listed above in detail below:

Figure 26 Input image of 7 segment real world test
**Shakiness**

Upon reviewing the recorded video, I noticed that there were a lot of vibrations in the footage, due to vibration of the floor caused by athletes running on the floor. Such shakiness caused the green dots that marks the location of the 7 segments on the 7 segment display to no longer stay at the correct position. Therefore, a wrong or incorrect output was obtained.

Figure 27 illustrates this issue. The green dots are the location of the 7-segment that the user clicked. Vibration causes the dots to no longer read the correct on-off status of the segment. An example would be the digit “26” on the first row. The dots labelled “a”, “b”, “c”, “d”, should read the segments as “on”. However, due to vibration, “off” was recognized.

The result is a 7 segment that does not represent any number as illustrated in Figure 28 below. Since it does not resemble any number, the program will recognize it as an “invalid” input.

![Figure 27 Illustration of dots reading wrong on-off status due to vibration](image)

![Figure 28 Incorrect recognition of number due to vibration](image)
Obstruction

The obstruction detection algorithm above also proved that it does not work in real life. The obstruction detection assumed that the obstruction will not be of a color that is similar to the scoreboard, i.e. black. However, some athletes were wearing sportswear that were mainly black that day. Therefore, the obstruction detection did not work.

Figure 29 shows the thresholded image with obstruction detection. The two dots illustrate the an arbitrary location that the user clicked on that should be black throughout the whole race. However, since a large proportion of the athlete’s body also became black after thresholding, and the location of the two green dots is indeed black, the obstruction detection failed to detect the obstruction. As it fulfilled the criteria of:

If there is no obstruction, the two dots must always be black.

Therefore, it is clear that another method of detecting obstruction is needed.

Since this 7-segment detection algorithm assumes that the scoreboard and the camera does not move, and the testing above shows that this assumption is not reasonable in real world applications. Therefore, I used another approach of using a 7-segment language pack for Tesseract to carry out the recognition of 7-segment numbers at last.

Second approach of carrying out OCR on 7 segment scoreboard

My second approach of carrying out OCR on 7-segment scoreboard was to use Tesseract. A user on GitHub trained a Tesseract language pack with 7-segment numbers.

The Github repository Shreeshrii/tessdata_ssd had 3 versions. I used the “fast/integer version”, language name “ssd_int”, as speed was one of my goals in this project.

I re-ran the test with the 7-segment language pack, and again set a whitelist of numerical characters only (0-9) in Tesseract, as an attempt to increase the accuracy of Tesseract output.
The results were very promising, much better than using a regular language pack.

Figure 30 shows the OCR recognition results using the “ssd_int” language pack for Tesseract on a pre-processed scoreboard image. The blue rectangles show the regions of interest marked by the user, and the green texts shows the OCR recognition results, with the smaller number % as the confidence. We can see that Tesseract, when used with the ssd_int language pack, was able to accurately correctly identify the 7-segment digits with high confidence of approximately 96%. This makes this approach the best solution for recognizing digits on a 7-segment display.

Figure 30 7-segment OCR result on 7-segment scoreboard with Tesseract ssd_int language pack
However, the ssd_int language pack outputs incorrect results when the scoreboard is partially obstructed as illustrated in Figure 31 below.

When the scoreboard is partially obstructed, Tesseract with ssd_int recognized the number “74” in the region of interest 2 (top right corner), with a confidence of 61%. Since it is impossible that the scoreboard cannot be obstructed for this program, obstruction detection or OCR results filtering has to be in place. As described in sections above, a generalized obstruction handling is adopted. Since there is a huge difference in the confidence values of the correct and incorrect (partially obstructed) OCR results, (95% and above in unobstructed regions of interest, and approximately below 75% in obstructed regions of interest), the results acceptance confidence threshold can be adjusted to a rather high 90%. This allows the incorrect results to be filtered out easily without referencing the previous frames.

As a result, the OCR component was able to identify the numbers on the scoreboard discussed above with very high accuracy.

The test was carried out with a scoreboard video that is affected by vibrations. In 3328 frames analyzed (25 frames per second, 133 seconds), 121 non-repeating OCR results (stabilized OCR results) were accepted. There were 2 errors in 121 results, therefore the accuracy of the OCR component reached a high 98.3%.

The demo video can be accessed at [https://youtu.be/KIMHdex18X0](https://youtu.be/KIMHdex18X0).

The two errors were caused by the obstruction staying for too long in front of the scoreboard. Therefore, more frames can be considered in later versions of the OCR component to reduce the effect of temporary obstructions on OCR results.
Solving the challenge - accuracy of Tesseract:

When I began the project, I had a lot of issues with trying to get Tesseract to recognize the pre-processed scoreboard images. Images such as the different number “7” of different resolutions in Figure 32 below, could not be recognized.

The problematic images looked good to a human eye, however Tesseract may not be able to recognize it.

After a lot of trial and error and research, I ultimately found that the problem was due to the resolution of the input image. Upon resizing the image to a height of approximately 40 pixels in height, Tesseract was able to recognize the number. 

![Figure 32 Example of unrecognizable numbers](image)

Tesseract also had issues recognizing the text at times at the beginning. For example, it may not recognize text at all, even when the image is fairly clean as illustrated in Figure 32 above. Upon doing more research, I found that I should specify the Page Segmentation Mode (PSM) in Tesseract to help tesseract understand whether the image contains a single character, a single line of text, or multiple lines of text, etc. Specifying the PSM improved the success rate of Tesseract recognizing the numbers significantly.

Moreover, I also ran into the problem where Tesseract recognizes alphabets when the input image only contains numbers. To resolve this issue, I looked into the documentation of Tesseract, and passed a list of whitelisted characters to Tesseract. This also helped to increase the correctness of the output text.

Solving the challenge – Deskewing the image

OpenCV provides a method for deskewing an image. Typical applications that I found according to references from the internet includes deskewing a sheet of A4 paper that is taken at a slight angle.

The application is as follows: First, `cv2.minAreaRect()` is called on a pre-processed image with sufficient contrast, this function returns the 4 corners of the target area, e.g. a sheet of paper placed on a table. Then, `cv2.getPerspectiveTransform()` is called to compute the transformation matrix that is needed to transform the skewed image to a “flattened” image. Finally, `cv2.warpPerspective()` is called to apply the transformation to the input image, using the transformation matrix produced from `cv2.getPerspectiveTransform()`.

However, I soon found a problem when applying such methodology to deskew the scoreboards. Since Tesseract is sensitive to distortion (distortion must be minimized as much as possible to improve the accuracy of Tesseract), the distortion must be carried out correctly. Moreover, since there is no guarantee that there will be sufficient contrast between the scoreboard and the background for `cv2.minAreaRect()` to run well every time. Combined with the fact that the scoreboard and the camera are not likely to be moved in real applications, I decided to let the user...
manually pick the 4 corners of the scoreboard to ensure accurate deskewing. This saved the challenge and work of training a computer vision model that is able to identify all the different types of scoreboards accurately.

Accurate corner pinning of the scoreboard is critical, as the remaining image processing, and accurate OCR results, all rely on a well deskewed image. If corner pinning was not done properly, it will cause Tesseract to be unable to recognize the characters correctly.

Some tests were carried out with a paper cutout of a simulated scoreboard, and a webcam on a PC. The results are as follows:

When the scoreboard is only slightly distorted (Figure 33), combined with the 4 user specified green dots at the corners, the deskewed output (Figure 34) is deskewed reasonably well, and deskewed well enough that does not pose a problem for Tesseract to recognize.
However, when the input image is distorted more, the automatic matrix calculation from `cv2.getPerspectiveTransform()` described above is unable to deskew the image properly:

Figure 35 shows the input image, with the simulated scoreboard rotated approximately 25 degrees, and the 4 user specified green dots that denote the corners of the scoreboard to be used for deskewing. The output image (Figure 36) has the aspect ratio totally wrong. The original aspect ratio of the black rectangle is 16:9. However, the desktewed output image (Figure 36) has an aspect ratio of approximately 4:3. Since the text is heavily distorted, Tesseract is unable to recognize it well.

Therefore, I concluded that the deskewing process cannot be done fully automatically using OpenCV functions. Without information on the original aspect ratio of the scoreboard, it is impossible for the program to deskew the image properly every time, as it does not know what the “correct” output is. Moreover, scoreboards come in different sizes and shapes. Therefore, it is not suitable to hard code an aspect ratio for all scoreboards in the program. Instead, the program will first try to guess the aspect ratio of the scoreboard based on the location of the 4 corners of the scoreboard. Then, the user can adjust the level of deskew manually if the automatic deskew is incorrect. A combined approach was taken as the goal is to have the accuracy of the OCR output as high as possible.

![Figure 35 Input image of heavily distorted scoreboard](image)
Therefore, the scoreboard OCR component of the project is designed to have an option of using manual input, in the case where automatic deskewing failed to produce a good result.

The user flow for the deskewing part of the OCR component is as follows:

Given an input image of a skewed scoreboard (Figure 37).
**Step 1** – user manually pick the 4 corners of the scoreboard (Figure 38).

The green dots located at the corners of the black simulated scoreboard are the corners that the user specified.

![Figure 38 User specified corners](image)

**Step 2** – The system shows an automatically deskewed image that was computed using `cv2.minAreaRect()` function in **OpenCV**. The automatic deskew feature should work well in most cases. Unless the camera was placed at a very slanted angle from the scoreboard, or the scoreboard is tilted heavily away from the camera. In the case where the program was unable to deskew the image properly automatically, the user can use the slider to modify the “wideness” of the deskewed image.

Figure 39 shows the deskewed image output at various width setting. The window titled “Adjust aspect ratio” has a slider that allows the user to adjust the width of the image. The lower the value, the wider the deskewed image is. This allows the program to obtain a well deskewed image for OCR.
Figure 39: Deskewed output of various width setting

Through partial manual intervention of the “scoreboard corner locating” and “wideness adjustment” process, we are able to obtain a well deskewed image of sufficient quality for Tesseract to reliably and accurately run OCR on.
Challenge – Generalized obstruction detection

The topic of obstruction detection was discussed earlier in the section OCR Results processing and sending to server. However, the conclusion of using considering the past OCR results that passes the confidence threshold was not the first attempt I made to handle errors in OCR caused by obstructions.

In this section, I will discuss the different attempts I made in developing a generalized obstruction detection method.

Attempt 1 – Feature matching

My first attempt to achieve a generalized obstruction detection mechanism was through feature matching. In this approach, feature matching is used to detect whether the scoreboard is present in the image.

Feature matching requires an image of the scoreboard, and will try to match the scoreboard video feed through attempting to match the extracted features.

In feature matching, the scoreboard image, and video image, is first converted to greyscale using the function cv2.cvtColor().

Then, a SIFT object is created using cv2.SIFT_create(). The SIFT object is used to find the features - keypoints and descriptors in the two grayscale images.

Using the found keypoints and descriptors, a BF matcher is created using cv2.BFMatcher(), and matched using bf.match().

The matches with the shortest distance are drawn.

My attempt with a 7-segment digital scoreboard suggested that it did not work properly. Figure 40 below shows that most of the features that were matched came from the logo “SEIKO”. Few features were extracted from the main scoreboard area, as the scoreboard was mostly a rather “plain” image, with not much features for the feature extractor to extract.

Investigating the matched lines, it is also obvious that some of the matches plotted in Figure 40 were from the 7-segment numbers. However, the numbers change over the course of the event. This suggests that there is no guarantee that this feature matching approach will be able to consistently deduce the presence of the scoreboard over time, especially when there is no logo (brand name) on the scoreboard.

Moreover, an unobstructed view of the scoreboard needs to be obtained in the first place, before feature matching can try to match the existence of the scoreboard. This may not be possible in real life applications at times; therefore, this approach was not taken.
Figure 40 Feature matching attempt with a 7-segment digital scoreboard

Attempt 2 – Color difference
Color difference computes the color similarity at specified regions in the image. For example, the user specifies a certain region on the scoreboard does not change color throughout the whole event. An example of region that does not change color is illustrated in the blue rectangles in Figure 41 below.

The color difference between the “input video” and the “unobstructed view of the scoreboard taken at the same angle and framing as the input video” at unobstructed at the areas the user specified is computed. If the color difference is larger than the threshold, then there is an obstruction. For example, if red is found in the areas that should be black, then it suggest that there is an obstruction (illustrated in Figure 42)

Figure 41 Regions that can potentially be used for color difference approach of generalized scoreboard obstruction detection
Figure 42 Example of successful obstruction detection with color difference

Figure 43 illustrates a situation where color difference fails to detect the obstruction. Since the obstruction is dark grey, and is similar to the original color of the areas where color difference is computed. Due to the high similarity in color, obstruction detection through color difference will not detect the obstruction in this case. If an athlete wears clothes that have similar color as the scoreboard, and walks in front of the scoreboard, obstruction detection will fail to detect it.

Therefore, another approach had to be taken.
Solving the challenge – performance of Tesseract API

Tesseract is an independent executable. For the python program for the OCR component to be able to integrate and interact with Tesseract, an API needs to be used. The API acts as the bridge between the Python program and Tesseract executable. There are two commonly used Tesseract API for Python, they are pytesseract, and tesserocr.

I first developed the OCR component using pytesseract, as it was the most commonly used library among the tutorials and documentations regarding the use of OpenCV and Tesseract in Python that I read.

On an AMD Ryzen 3700u laptop, the OCR performance is approximately 1 fps on a given image of 360 by 640 pixels. I was not satisfied with this performance at all. Therefore, I carried out a profiling on the python program using cProfile. Through the use of cProfile, I found that there were a lot of calls to pytesseract.py(image_to_data) function, and the cumulative time was the highest. Therefore, I was able to confirm that pytesseract was the cause of the poor performance in my program, but not the image pre-processing done in OpenCV.

However, I soon discovered that the performance of pytesseract was very poor. Therefore, I looked into the source code of pytesseract.

pytesseract.image_to_data() is a function that is called for running OCR recognition on a given image. The function returns the 3 main information:

1. Tesseract OCR result
2. Location of bounding boxes
3. Result confidence

Therefore, this function is called once, every time an image needs to be recognized. Due to the nature of my program design, this function is called repeatedly in a while True loop.

Upon inspection, I discovered that when the function was called, it will first carry out a Tesseract version check. If the version check passes, it will start Tesseract, pass the image to Tesseract, parse the Tesseract output results, and finally return the Tesseract OCR results. The function can be summarized in the flowchart below (Figure 45). The flowchart is simplified, some details omitted.

The flowchart in Figure 45 is separated into 3 columns. The leftmost column are actions by my python program (in blue boxes), the center column in yellow boxes are actions in pytesseract API, while the rightmost column in green boxes are actions done by the Tesseract executable.

When the function pytesseract.image_to_data() is called, pytesseract API first checks the version by running “tesseract.exe –version” via the subprocess.Popen function in Python. If the installed Tesseract version passes the version check, pytesseract calls the Tesseract executable with the image stored as a file, and other configuration options by subprocess.Popen. After the image is recognized, Tesseract will return with the OCR results, and pytesseract will parse the output, and return.

I initially suspected that the version check was the main cause of the long return time by pytesseract.image_to_data(). However, upon removing the version check code (lines 521 to 522 in pytesseract.py, shown in Figure 44), test showed that there is no significant improvements to the time taken for the pytesseract.image_to_data() function to return.
Therefore, I did more research and investigated into the pytesseract source code. I identified the problem to be with pytesseract's implementation of calling the Tesseract.

Every time I need to run OCR on an image, the function pytesseract.image_to_data() is called. pytesseract.image_to_data() starts a new instance of Tesseract, Tesseract runs recognition on the input image, and the Tesseract instance is destroyed right after the image OCR is completed. This behavior is demonstrated in the rightmost column, green boxes in Figure 45.

I identified that the main cause of this slow response time was due to the startup time of Tesseract. Since reducing the input image size did not yield any significant improvements in speed.

Therefore, I switched to another Tesseract API for Python: tesserocr.

Tesserocr runs faster than pytesseract in my use case that needs to run OCR on different input images continuously, as it allows the reuse of the Tesseract instance. Therefore, the slow
startup time of Tesseract only affects the OCR time of the first image. OCR on subsequent frames, or images, will reuse the same Tesseract instance, without a cold startup of the Tesseract executable.

This is illustrated in Figure 46 Comparison between the two Python tesseract APIs, pytesseract (left) and tesserocr (right) when multiple frames need to be recognized. The left half of the diagram shows the program calls when pytesseract is used. Every time an image needs to be recognized regardless of whether it is the first, or second and subsequent frames, pytesseract will start a new instance of Tesseract (green box with red outline).

The right half of the diagram illustrates what happens when tesserocr is used instead of pytesseract. The “Tesseract Startup” process (green box) only occurs once in the whole recognition process. After recognizing the first input image, the Tesseract executable will not exit. It will be kept to be reused in second and subsequent frames.

Therefore, it improves the performance of the program significantly.

Although Tesserocr runs much faster, it comes with its limitations. Tesserocr does not provide as much high level functions compared to pytesseract. To obtain the same output as pytesseract’s pytesseract.image_to_data() function, I have to run the following commands:

1. Api.SetImage(inputImageObject)  #provides the input image to Tesseract
2. api.GetComponentImages()  #find the bounding boxes
3. api.GetUTF8Text()  #gets the recognized text
4. api.MeanTextConf()  #returns the confidence level of the recognized text

Through changing the API from pytesseract to tesserocr, by reusing the Tesseract instance, I successfully achieved a frame rate of approximately 30 frames per second (if there’s only one ROI in the input image), effectively a 3000% improvement in performance.

When the number of ROIs in the image increases, the performance gain also increases by a similar magnitude. 4 ROIs in the image with pytesseract has a performance of approximately 0.25fps. However, with tesserocr, the performance still maintains at approximately 20 frames per second, which is an 8000% improvement in performance.

Therefore, tesserocr was the Tesseract API that was used for the OCR component of this project.
Figure 45 A simplified flowchart that shows what is being done when the function `pytesseract.image_to_data()` is called in `pytesseract API`
Figure 46 Comparison between the two Python tesseract APIs, pytesseract (left) and tesserocr (right) when multiple frames need to be recognized.
Challenge – Fonts of scoreboard

The fonts of certain manual scoreboards used in sports have some fairly special fonts that are not commonly used in everyday life. One characteristic is that the font is stretched along the y axis, and the shape of the numbers also differs from what we commonly see.

The discussion below is based on my research of manual flip-page style scoreboard on the reseller websites on the internet.

An example is illustrated in Figure 47 below. When we compare it to the more commonly used fonts such as Calibri, or Times New Roman in Figure 48, the number “0” is significantly longer than regular fonts. Moreover, the number “0” has a straight vertical line on left and right sides, and the curve is only round at the top and bottom. This is different from a regular “0”, where the text is not as long, and the left and right sides of the “0” do not resemble a vertical straight line.

Figure 47 Photo of Manual flip page scoreboard showing 000 0 9 999

Figure 47 Source: https://detail.tmall.com/item.htm?abbucket=14&id=622716367333&ns=1&skuId=4572445407799&spm=a230r.1.14.16.44401379AnMtdv

Figure 48 Numbers 000 0 9 999 in Calibri (top) and Times New Roman (bottom)

The same observation can be made in Figure 49. The wideness of the number “0” at the bottommost scoreboard, compared to the number “31” on the bottom-most scoreboard, is very different. This poses a challenge for Tesseract to identify, as the recognition result of Tesseract is easily affected by the level of distortion of the character.

Figure 49 Image from online reseller that shows 3 scoreboards of different styles

Figure 49 image from https://item.taobao.com/item.htm?spm=a230r.1.14.34.44401379AnMtdv&id=691687989083&ns=1&abbucket=14#detail
The same observation can also be made to the scoreboard in Figure 50. The number “0” is a complete straight line on the left and right sides, and the curve is only round at the top and bottom. Which is quite different from a normally typed “0”.

![Figure 50 Image of scoreboard from online retailer website showing the numbers 00 00](image)

This poses a challenge for Tesseract, as Tesseract does not seem to be trained to recognize those fonts. In the situations where the font is recognized, the confidence level is lower than a regularly typed text.

Therefore, a limitation of the current scoreboard OCR system is that it may not recognize certain scoreboards with specific fonts, until the OCR engine model is trained specifically on scoreboard fonts.

**Performance**
In my testing on a laptop with AMD Ryzen 3700U, a frame rate of 10 to 30 fps is achieved, depending on the number of ROIs specified. Considering that the scoreboard is only updated at a frequency of 1Hz (in the case of the “second” unit in the time). In the case of scores for sports matches, the scores are updated with intervals of at least 10 seconds in-between, depending on the sport. Therefore, the performance that I achieved on a laptop with average processing power is more than sufficient.

**Final testing result of the end product on scoreboards**

**Conclusion and Future Works**

**Conclusion**
To conclude, I completed the system design of an athletics information system, and completed the OCR component of the proposed system with a generalized scoreboard OCR software that can be used on different types of scoreboards.

Achieving the high accuracy and high performance on the OCR component was not an easy task, and a lot of optimizations on image pre-processing had to be done through extensive research and testing to achieve the high accuracy of >98% in my testing. As shown above, without image pre-processing, Tesseract was unable to identify the numbers most of the time. However, after pre-processing the image through multiple steps including deskewing, greyscaling, binarization, erosion, dilation, cropping to region of interest, resizing, thresholding, and padding borders, I was able to achieve a high accuracy and high performance OCR system given that the font is commonly used.

There are certainly some limitations to the system that can be improved, including poor accuracy when non-common fonts are used, and some manual input is currently needed, which will be discussed below.
Future work – Retraining tesseract on manual scoreboard fonts
In my testing with Tesseract in real world with some manual page-flip scoreboards, and my research on manual flip-page scoreboard that are commercially available on the internet, I observed that this type of scoreboard often uses a fairly special font.

Therefore, I believe that I should retrain Tesseract on these fonts that are commonly used in manual style scoreboards. Alternatively, I can train my own CNN model using packages such as pytorch in Python, and use a combined approach of Tesseract and pytorch to deduce the OCR result to achieve a higher accuracy. For example, if Tesseract is unable to recognize the number with 90% confidence, while my model can identify the model with high confidence, then the recognition result from my own model should be accepted.

Future work – Auto-scoreboard discovery
Currently, the scoreboard area is manually specified by the user specifying the 4 corners of the scoreboard. In the future, this process can be automated by training a generalized scoreboard detection computer vision model to automatically identify the location of the scoreboard.

This can allow the system to continue to function on the rare occasions that the scoreboard and/or the camera changes position.

If the model performs well enough it might not even be necessary to have a dedicated scoreboard camera for OCR recognition. A wide angle camera, or a moving camera, may be used for scoreboard OCR, where the program will automatically find, and extract the scoreboard in the video for recognition.

Future work – internally generate more versions of the image for recognition
In my testing, I found that an incorrect, or non-ideal deskewing, non-ideal rotation, or non-ideal erosion/dilation setting may cause Tesseract to be unable recognize the text properly. Therefore, I propose that the program can internally generate different versions of the cropped ROI image to be passed to Tesseract, or the OCR engine.

For example, instead of solely passing one image for Tesseract, the program should also rotate the cropped ROI image slightly, and pass it to Tesseract. The result with the highest confidence will be accepted as the “correct” OCR result.

This should make the OCR system more robust, and prevent slight user misconfigurations from leading to incorrect or no OCR results.

Future work – Improving OCR correctness based on race logic
The numbers on scoreboards follow certain logic. For example, the “score” field in most sports e.g. handball, football, volleyball, should only go up, in increments of 1. This “increment by 1” characteristic can be used to filter the OCR results. If a number is identified to the scoreboard that was not incremented by 1 compared to the last number in the same position, it is very likely that the OCR result was wrong.

In the case of scoreboard OCR for athletics field events, it the value “distance” is only reasonable in a certain range. For example, the distance of long jump athletes is typically between 1 to 13 meters. Therefore, any result that is beyond this range can be considered as incorrect.
When no valid OCR results is recognized, an alert can be made to the program user, and prompt the user to confirm whether the OCR results are correct, or input the correct result, or reconfigure the OCR setup if needed.

**Future work – complete the rest of the information system**
Due to the limited time on this project, I was unable to complete the whole information system. Therefore, completing the whole information system would be a future work.
Works Cited


