Validación automática de guías de accesibilidad en videojuegos
Automatic validation of accesibility guidelines in videogames

Trabajo de Fin de Grado
Curso 2021–2022

Autor
Adrián Álvarez Bernabé
Esteban Restrepo Gutiérrez

Director
Guillermo Jimenez Díaz

Colaborador
Alessandro Lentini

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Validación automática de guías de accesibilidad en videojuegos
Automatic validation of accesibility guidelines in videogames

Trabajo de Fin de Grado en Desarrollo de Videojuegos

Autor
Adrián Álvarez Bernabé
Esteban Restrepo Gutiérrez

Director
Guillermo Jimenez Díaz

Colaborador
Alessandro Lentini

Convocatoria: *Junio 2022*
Calificación: *Nota*

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

30 de mayo de 2022

Resumen

La accesibilidad es un aspecto fundamental de los videojueogos. Todos los jugadores, independientemente de sus capacidades fisicas, deben poder disfrutar de ellos. Alrededor del mundo, distintos gobiernos y asociaciones han creado estándares con los que medir la accesibilidad de los videojuegos para que puedan ser disfrutados por todos los públicos. Uno de estos casos que hay que tener en cuenta al desarrollar videojuegos, y en el cual se centra este trabajo, es el de las personas con discapacidad visual como el daltonismo o la miopía. Ellos pueden tener dificultad para ver bien el texto en un juego cuando este es muy pequeño en relación al tamaño de la pantalla o cuando no hay suficiente contraste de color entre la letra y el fondo.

A dia de hoy es comun que las empresas cuenten con una extensa plantilla de *testers* que comprueba manualmente los requerimientos legales y la calidad del juego. Estas comprobaciones, aunque sencillas, resultan laboriosas y consumen un tiempo muy valioso que podría dedicarse a pruebas sólo realizables por humanos. Además, la tendencia a crear juegos que tienen cada vez más contenido hace que este sea un volumen de trabajo cada vez menos asumible por una fuerza de trabajo manual.

Este trabajo consiste en el desarrollo una herramienta que permite automatizar el reconocimiento de texto para detectar su tamaño y su contraste con el fondo, para asegurarnos de que cumple una serie de criterios que aseguran su accesibilidad a varios públicos.

Palabras clave

accesibilidad, videojuegos, texto, contraste, luminancia

Abstract

Accessibility is a fundamental aspect of video games. Any player should be able to enjoy a game, independently of their physical ability. All around the world, governments and associations have implemented standards for game developers to make sure their video games can be enjoyed by everyone. One of the cases to keep in mind when developing games, and the one this project is based on, is the case of people with deficient color vision or nearsightedness. These users can have difficulties seeing text in-game when it is relatively small when compared to screen size or when its color does not have enough contrast with the background.

It is commonplace for companies to have a considerable team of testers who manually check for legal accessibility requirements as well as ensuring the quality of the product. These checks, although simple, are time consuming, taking up valuable time that could be used for more complex checks that could only be made by humans. Moreover, tendency to create increasingly content-packed games creates less manageable workloads for manual checks.

For this project, a tool has been developed that automates the text detection and calculates its size and relative luminance contrast with the background. This tool is used to verify that a series of accessibility criteria are being met.

Keywords

accessibility, video games, accessibility, text, contrast, luminance

Contents

List of figures

Chapter 1

Introduction

Somebody has to start. Somebody has to step forward and do what is right, because it is right.

Kaladin, The way of Kings, Brandon Sanderson

This chapter will go over the motivations, objectives and work plan for our tool TinEye. TinEye will be a tool for automatic validation of accessibility guidelines. It will ensure that people with vision deficiency can enjoy games the same way other people do by recognizing text in a game and making sure it passes certain accessibility requirements.

1.1. Motivation

It is estimated that 5%-8% of men and 0.85% of women suffer from some type of color vision deficiency (Tanaka et al., 2010) and sight impairments such as myopia are on the rise (Fredrick, 2002). The implications of deficient color vision are specially important when dealing with text. Color blindness and myopia do not only conflict with the users perception of the artistic media, they can be detrimental to the point of not noticing critical information when the text lacks sufficient size and contrast with its background.

Any player should be able to enjoy a game, independently of their physical capabilities. As the the importance and relevance of video games increases in our society, governments and companies are taking action. Currently the responsibility falls on game development studios to make games as accessible as possible, but some countries like the United States are enforcing laws that could sanction studios that do not take into consideration the accessibility aspect of their titles (FCC, 2010).

An example of one of these laws is that for communication between players, such as in-game chat, the text has to comply with minimum set sizes to ensure that most users will be able to read it. This forces studios to spend extra resources on just making sure that the legal part of accessibility is being met instead of being able to focus on the players themselves. At the time of development, tests like the one just mentioned are executed by humans manually measuring words and contrast with image processing software. This makes the process slow and prone to human error.

The video game market continues to expand as it maintains the steady growth presented during the last decade (Giuditta de prato, 2014). While the market grows, so does the size and quantity of games published every year, making this manual testing less sustainable

as more testers are needed each time and small studios cannot dedicate as much time to ensuring every in-game appearance of text is accessible.

There is a tendency in recent games to offer as much content as possible. While this can be beneficial to players, the amount of testing needed to ensure a quality of more ambitious products grows exponentially. There is a need to free testers from manual, tedious works and letting them test things only humans can do.

1.2. Objectives

The goal is the development of a tool that identifies text that is present on screen and analyzes it to make sure that users with deficient color vision or certain forms of sight impairment can enjoy the game as well. The tool will check for requirements that include font size checking for users with myopia and contrast checks for users with deficient color vision. The parameters for these checks must be easily interchangeable since requirements in these field are still being determined and can change at a moment's notice.

The project will be developed as a tool for Electronic Arts that will improve the accessibility of their games as well as increasing the employee productivity. There is no public information about similar tools that automate this process, which means the approach will focus on understanding the manual process and adapting it to a workflow that allows it to be automated.

1.3. Work plan

The methodology used during development will be Scrum, an agile methodology focused in short iterations called *sprints* that always end with a working version of the product, ensuring nothing breaks during developments and there's always something to show for progress. This methodology helps the inclusion of feedback in early stages of development and keeps the project adaptable and open to change. The sprint duration is in the form of two-week increments, at the start of which the most important tasks (or *user stories*) are chosen and divided among the team. During the sprints, the stories are completed and new stories are added to the backlog. Once a sprint is finished, the most important stories that remain in the backlog are chosen for the following sprint.

The first sprints will be focused on the study of Electronic Arts' current process for validating video game accessibility and prototyping ways of automating the core steps of this process. This investigation will also need to assess the current state of media accessibility and requirements and guidelines throughout the video game and other media industries to determine which changes can occur regarding accessibility guidelines in order to make the tool easily configurable and ready to take on new changes as guidelines evolve.

Once the state of the art has been sufficiently investigated, a design outline for the tool will be devised; describing its desired operation, the analysis we want it to do and how it will be developed. A series of prototypes will be made to ensure that the proposed approach to develop the tool is viable and works reliably.

When sufficient confidence in the viability of the prototypes has been achieved the development of the tool will shift away from prototypes and towards a library that can handle different types of media and can output a report that qualifies the media as compliant or not compliant with the guidelines specified.

Development will be split into three stages: first the detection of the existence and position of text, the measurement of this found text to comply with size requirements and

finally the contrast measurement of the text with its background to assess compliance with contrast requirements.

Nearing the end of the development of the tool, a series of tests will be conducted. Various teams from different departments and studios at EA will provide real world test cases where a set of media would need to be compliant and will be processed by the tool. The reports provided will help to determine the capabilities, limitations and possible bugs of the tool and its projected impact in the current workflow for accessibility validation.

1.4. Document outline

The memory for this project is structured as follows:

1. Introduction. The objectives and the motivations for the project are outlined.
2. State of the art. Explanation of different vision impairments that affect video game users and how they change their experience, the standards followed in the industry regarding accessibility, and possible tools to improve on these guidelines.
3. Design. A description of how the tool developed in this project would function, its features, the project structure, the architecture, and its tests.
4. Development. The methodology followed during the development of the project as well as all of the steps taken while developing it and the reasoning behind them.
5. Evaluation and discussion. Results and analysis of diverse tests and benchmarks ran on the tool to ensure it works correctly and in a desired manner, highlighting the advantages and limitations of the project.
6. Conclusions and future work. Discussion of the results obtained from this project and possible future work and investigations that could sprout from it.

Chapter 2

State of the Art

This chapter will detail some of the different vision impairments that video game users suffer, like color vision deficiency (section 2.1) or myopia (section 2.2), and how they can affect their game experience. Next it will mention different standards and guidelines that are used in the industry to ensure any person can enjoy in section 2.3, and how they have to take into account topics like color spaces (section 2.4) since they change how colors are represented on a screen. Finally, various text detection and recognition solutions that are available and could be used to detect screen text will be explained along with their most common use cases in section 2.5.

2.1. Deficient color vision

Deficient color vision, inaccurately called *color-blindness*, occurs when someone cannot perceive color as the majority of people. This situation emerges from the partial malfunction of a group of color cones in the eye. Humans perceive color from three types of cones in the eye, sensible to the different wavelengths of the light spectrum attributable to red, green and blue (Sharpe, 1999). When one of these types of cones is missing or malfunctioning, it becomes difficult or even impossible to distinguish between some colors. Depending on which types of cones are impaired, the different cases receive different names:

- Protanopia: no red cones
- Protanomaly: impaired red cones, can perceive some shades of red
- Deuteranopia: no green cones
- Deuteranomaly: impaired green cones, can perceive some shades of green
- Tritanopia: no blue cones
- Tritanomaly: impaired blue cones, can perceive some shades of blue
- Achromatopsia: complete lack of color vision.

Deuteranopia, protanopia and tritanopia are grouped as dichromacy. (Tanaka et al., 2010). Comparison of how a person with each type of color vision deficiency would see the standard color wheel can be seen in figure 2.1

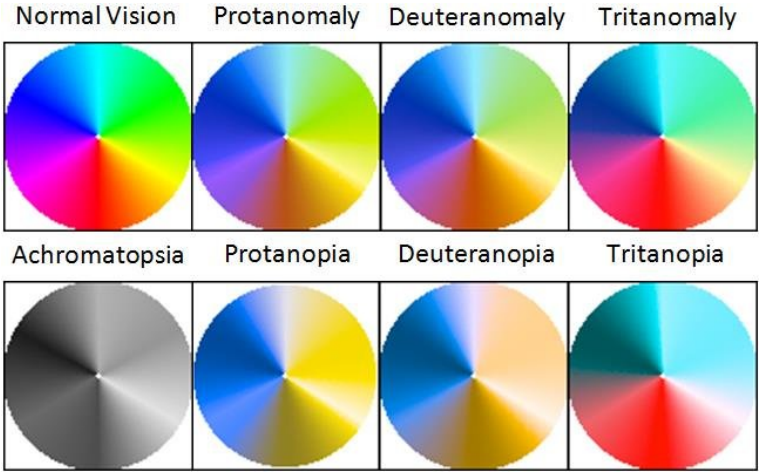


Figure 2.1: Comparison of the visible color spectrum in common types of color vision deficiency (Melillo et al., 2017)

2.1.1. Making media accessible to color vision deficiency

There are many valid approaches for making media accessible for people with deficient color vision. One of the methods to improve legibility is changing the color lightness (Tanaka et al., 2010), which involve small changes without completely altering the color. This makes for an increase in contrast that makes two previously similar colors distinguishable for people with deficient color vision. The problem with lightness modification is its cost, as most of this methods require various iterations, complex algorithm, or a set of parameters that vary from frame to frame.

Another method widely used for making media accessible is color filtering. By specifying the type of color deficiency, problematic color values for the specific deficiency can be swapped out for others so the user can enjoy the media as showcased in figure 2.2.

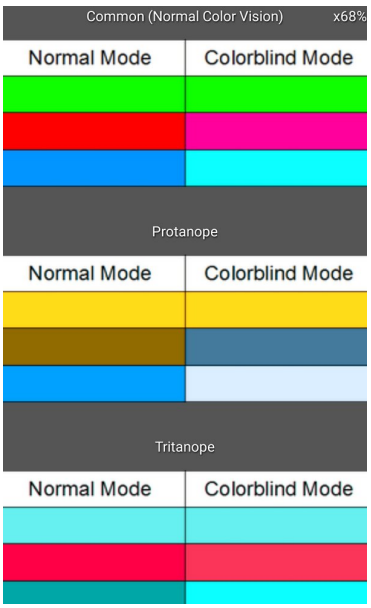


Figure 2.2: Comparison of how color filtering enables color vision deficient users to better distinguish certain colors (Tycho Henzen, 2021)

2.1.2. The case for videogames

Video games present some unique challenges in comparison to other media: what is displayed on the screen is not known beforehand like a movie, it is being generated and rendered up to sixty times a second.

Performance is a critical aspect in most video games for a smooth experience, particularly with the rise in popularity of virtual reality (VR); the high cost of the previously mentioned lightness modification makes it a non viable solution for solving color vision deficiency.

In addition to technical challenges, video games also face artistical ones. Color is as fundamental to videogame design as it is to any form of art. One of the main goals of players is to feel some sort of emotion (Joosten et al., 2010); artists then use every available tool, including color, to craft these emotional pieces. This raises a problem with color filtering: the intended vision of the designers may be mangled by color filtering, as seen in figure 2.3.

Figure 2.3: Game Overwatch with normal color settings (above) and with tritanopia filter applied (below) (Ashley Wood, 2017)

2.2. Myopia

Myopia, commonly referred to as nearsightedness, is a vision deficiency that causes the person to see objects that are far away as blurry while still being able to see closer objects clearly.

Myopia is greatly present across the world, with the highest incidence in asian popu-

lations (84% in 16 to 18 year olds) and with lesser but still significant incidence in other regions: 49.7% in Swedish kids aged 12 to 13, or 37.2% in 10 to 15 year olds in Greece and Bulgaria (Wu et al., 2016).

People with myopia commonly struggle to see far away text and images like road signs or billboards clearly. This may not seem too relevant for video games at first, but consoles (representing more than $50 billion in revenue in the global market in 2021 (Tom Wijman, 2021)) are recommended to be played from a distance of at least one meter (HDhes, 2014), which for higher diopter users already poses a problem for legibility.

The commonly accepted solution to this is ensuring that text has a minimum size depending on the screen size so that it is legible for most users. For example, Microsoft has implemented a set of guidelines that a games must follow in order to be published for their console.This means that an Xbox title's text has to have a minimum height of 28 pixels in a 1080 pixel tall screen (Microsoft, 2021).

2.3. Accessibility Guidelines

To ensure that any person regardless of their ability can enjoy video games and other media different governmental bodies and independent organizations around the world have created sets of guidelines for accessibility. This way users who have special needs are be able to completely enjoy a piece of media like the normative target audience would be. The following subsections will explore the main guidelines being followed.

2.3.1. Communications and Video Accessibility Act

The 21st Century Communications and Video Accessibility Act (CVAA) (FCC, 2010) comprises a set of legal requirements that media should fulfill in the United states, including videogames. This was created to to update the accessibility laws created in the 1980s and 1990s to modern digital standards.

This act enforces items like closed captioning availability being mandatory for any digital video distributed by companies in the United States. This requirement is applicable to videogames. Many videogame titles use pre-rendered footage known as cut-scenes for storytelling that are subject to this law.

CVAA sets a legal standard for all communications inside a game such as interconnected voice over protocol (VoIP), non-interconnected VoIP, electronic messaging, inter-operable video conferencing and video communications.

This law is applicable to media producers and videogame publishers with sufficient resources to enforce it. This presents a double edged-sword for videogame companies. The companies that have the resources to take action are usually the ones who offer bigger amounts of content, making their games harder to test.

2.3.2. Web Content Accessibility Guidelines

The World Wide Web Consortium (W3C) also created their own set of guidelines for digital content called the Web Content Accessibility Guidelines (WCAG). They "define how to make Web content more accessible to people with disabilities [...] including visual, auditory, physical, speech..." -W3C (2008b).

The web content accessibility guidelines have gone through various iterations, version 1.0 being released in 1999, 2.0 in 2008 and 3.0 in the works, with a working draft from 2021.

W3C currently recommends content creators on the internet to adhere to the WCAG 2 guidelines. WCAG 3.0 builds upon previous versions but its methods to ensure accessibility change, making some content that passes following the 2.X standards to not pass anymore, as will be detailed shortly.

2.3.2.1. WCAG 2.0

The WCAG 2 guidelines detail various techniques and requirements to ensure that content is not only accessible to people with color vision deficiency or lesser sight impairments like myopia but also to users with more limiting conditions like blindness or deafness.

The article that applies to our work from the WCAG 2 guidelines is 1.4.3: "The visual presentation of text and images of text has a contrast ratio of at least 4.5:1" (W3C, 2008a). It states that text should have a minimum luminance contrast ratio of 3 between text and background when certain size requirements are met or 4.5 when these size requirements cannot be ensured.

Luminance (L) is used to measure the intensity of light in a given area; measured in candela (the SI unity for measuring luminous intensity) per square meter. Relative luminance (Y), on the other hand, normalizes luminance values from 0 (absolute black, no light being reflected) to 1 (absolute white, perfect reflection) (Sik-Lányi, 2012).

Relative luminance is the standard unit of measurement when working with digital media because it allows developers to avoid thinking about how each color combination might be perceived by users and just keep in mind about how bright or dark a color will be seen when playing independently from the display.

The WCAG 2 standard (section 2.3.2.1) dictates that a contrast ratio of 4.5 or more between the background color and the text color ensures that most users will be able to distinguish the contents, regardless of the colors used. This is possible because, as exposed in section 2.1, luminance is perceived separately from color. This contrast ratio is calculated with equation 2.1.

$$Ratio = (L1 + 0.05)/(L2 + 0.05) \tag{2.1}$$

Where L1 is the relative luminance of the lighter color and L2 the darker one.

2.3.2.2. WCAG 3.0

WCAG3 updates the calculation for contrast checking in text based on more recent research on color perception. It uses the Advanced Perceptual Contrast Algorithm to calculate contrast (W3C, 2021). This algorithm, apart from the background and foreground color like the WCAG 2 calculations do, also takes into account the text size and weight relative to the screen space it occupies.

2.3.3. Microsoft guidelines

The tech giant Microsoft has also created its own accessibility guidelines for titles released on their Xbox systems. They detail the standard text display sizes and contrast requirements but also set standards for in-game wait times, UI navigation, error messages or haptic feedback (Microsoft, 2021).

Microsoft's text size guidelines state the following for text height in console titles:

- 26 pixel height at 1080p

- 52 pixel height at 4k

2.4. Color representation

As explained along WCAG 2.0 (section 2.3.2.1), it is important to make our calculations in a way that matches the physical perception of light as accurately as possible. The first measure was the calculation of relative luminance with the equation 2.1. Luminance allows us to measure how *'bright'* or *'dark'* a pixel is without having to take into consideration how its specific color is perceived. After it is assured that a color luminance is perceived equally by everyone, it is needed to assure that the color displayed is the same among all displays.

It is important to denote the existence of color spaces and their significance when working in this field: the human eye can see a wide range of colors present in the world, but when trying to present these colors through a screen, not every color the human eye can see can be faithfully represented by hardware, the more exact you want your colors to be in relation to reality the more expensive and difficult to come by the hardware for it becomes.

To work around this issue, color spaces were invented. They are standardized collections of colors that could be represented on screen so that the same color is ensured to be equal across other devices. Thanks to color spaces, when creating an image in one screen you would know it would show up the same on a different one if both are capable of showing the specified color space.

The comparison between human vision and different color spaces can be seen in image 2.4. As can be seen, pure green in sRGB does not equate to pure green in other color spaces. If we wanted to use this color value for luminance calculations we need its actual color value in 'human' color space; this transformation is known as linearization.

The standard color space for computers and web is sRGB, which is clearly more limited than ProPhoto RGB or Adobe RGB, but wider color spaces require more expensive hardware, which is the reason such a color space is as extended and wider ones are only used in professional settings where the most faithful representation of colors is needed.

2.5. Text detection and recognition

The automation of accessibility testing requires some form of text detection or recognition. In the following section, the most popular libraries and techniques will be discussed. Text detection will cover the detection of text inside an area, without actually being able to read its characters. On the other hand, text recognition will be able to tell what words are written in an image.

Optical Character Recognition, or OCR, is a technology that allows recognizing characters, such as letters and symbols, through an optical mechanism, such as an image taken of a piece of paper. Although much less capable than humans, this technology is widely used to identify, recognize and digitize text found in images like scans from books and extract their text (Mithe et al., 2013).

These technologies can be used to also identify text in digitally created images, like a screenshot of a scene in a video game.

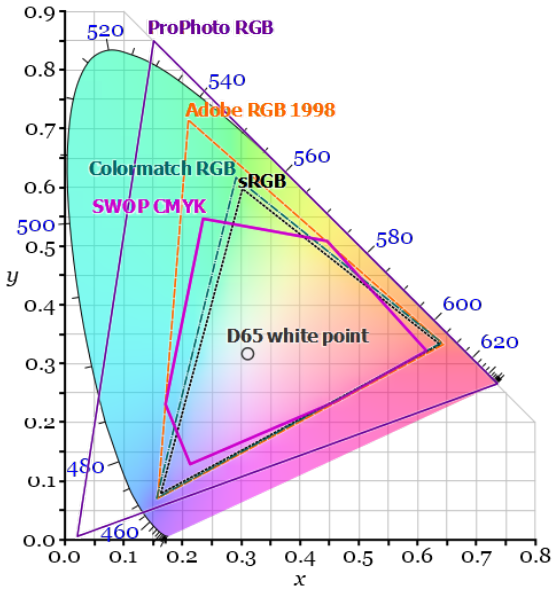


Figure 2.4: Color space comparisons (BenRG and cmglee, 2014)

2.5.1. EAST - Efficient and accurate scene text detector

For the purpose of development, the technology used for analysing text in real world scenarios will be researched. Videogames are a much more complex scenario for text detection than plain text: they include a variety of fonts, colors and complex backgrounds. Because of this, the advancements in scene text detection can provide very useful results for the development of the tool.

EAST, Efficient and Accurate Scente Text Detector, is a deep learning based on automatic feature detection of the different characters. Its development was motivated by increasing the F-score and speed of the current solutions. EAST approach tried to simplify the pipeline, removing many stages considered unnecessary and using two-stage pipeline (see figure 2.5). EAST's first stage is a fully convectional neural network that assigns a value ranging [0,1] to each pixels and then encloses the values in quadrangles. The second stage is a post-processing stage, the geometries that "survive" the thresholding stage are processed by a NMS, non-max suppression, algorithm. NMS ensures that there is a single quadrangle for every single piece of text that covers its whole area instead of various overlapping quadrangles. (Zhou et al., 2017)

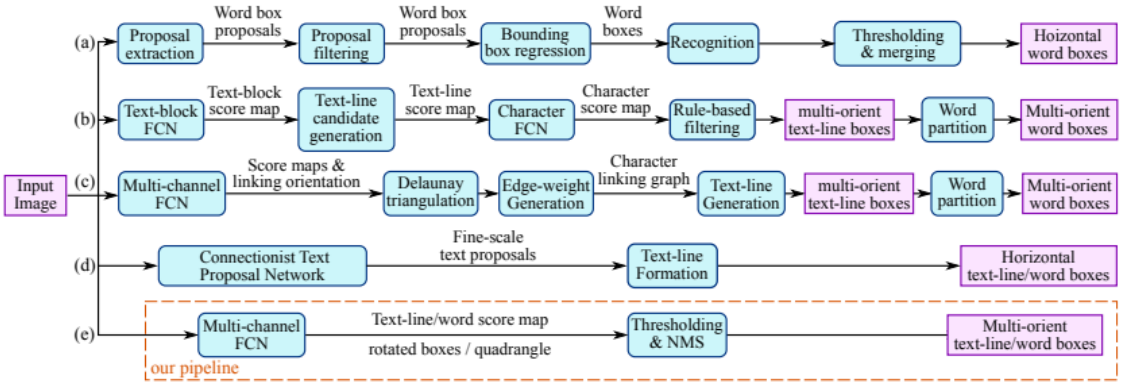


Figure 2.5: Comparison of different scene text detection algorithms from Zhou et al. (2017)

From our perspective of users, when using EAST as a black box we need to keep in mind its expected input and output. EAST input must be a 3-channel image whose dimensions are a multiple of 32, and its output is a series of quadrangles. This will make it necessary to keep in mind because the scale will need to be restored in order for the measurements to be accurate.

2.5.2. Tesseract - Optical character recognition engine

The most widespread solution for text recognition is Tesseract. Originally a Hewlett Packard product, it was released as open source in 2005 (Community, 2022). Tesseract has been developed mainly as a tool for digitizing documents; it has several layers of preprocessing that are not useful for digitally generated images, for example steps are taken to reduce wrinkles in paper or artifacts that appear when scanning a document. There is a focus on errors natural to text documents such as misaligned text, bent pages, broken characters (Smith, 2007) that do not apply when working with digital content.

When running an image through an OCR engine there is usually a necessary step taken called binarization, simplification of its pixels to a value of 1 or 0 so that it becomes an image composed of only black and white pixels to find out where the letters are. Tesseract can make most of the work for its users. First, it automatically binarizes the image being processed and then denoises it, finally running the text recognition part of the process. When working with digital images, easing this binarization is very important, as more complex backgrounds (opposed to just a white page or flat background from normal documents) are to be expected and the engine may not be as accurate.

2.5.3. OpenCV Text Recognition models

OpenCV offers high level APIs for both text detection and recognition. Text detection is useful for finding whether text is present in a scene and its position. Text recognition needs to be given a specific region of an image known to contain text outputs after obtaining these regions trough text detection. Afterwards, it gets the text contained within through a series of calculations using neural networks.

Operations with neural networks can be very efficient and seemingly magical, but they all depend on the neural network being correctly trained with enough examples in its training data sets.

The OpenCV documentation offers resources for various trained models, each one of varying complexity depending on how big an alphabet of possible characters it supports:

- CRNN for black and white images, 36 character alphabet (0-9 + a-z).
- CRNN for color images, 94 character alphabet (0-9 + a-z + A+Z + punctuations).
- CRNN for color images, 3944 character alphabet (0-9 + a-z + A-Z + chinese characters + special characters).

The availability of diverse pre-trained networks and the capacity of changing the recognized character set make OpenCV's text recognition a useful asset. A tool can be developed following only the English alphabet and, if needed, could later be expanded to include more alphabets. This system also offers more flexibility regarding its source media compared to Tesseract, which expects to get a scanned document.

2.6. Conclusion

This chapter has exposed how different sight problems can negatively impact players' ability to enjoy a game or read critical text. Mitigating this impacts is ultimately the developer's responsibility. Games should be universally accessible both because of ethical principles regarding its consumer base and because of the big percentage of players they can represent, like myopia in Asian populations.

The amount of resources needed to keep games accessible is considerable, and becomes bigger as the amount of content offered by games increase. While attention is brought to this topic, some countries like the United States are starting to enforce laws that all published media should comply. Console manufactures like Microsoft are also taking a stance towards accessibility. The console gaming business model relies heavily in what is considered *"couch gaming"*, being able to play in a living room far away from the screen, thus needing to ensure its readability. The inability to publish a game in a console due to accessibility guidelines could be a huge backlash for a publisher.

The increasing amount of resources needed to keep games accessible paired with the growing legal pressure makes the automation of these tasks a big asset for game development companies. Finally, the manual approach regarding many of these tests make them sub-optimal and prone to human error. Automation of this checks could result not only in time saved, but also in an accuracy increase.

Chapter 3

Design of a tool for automatically validating accessibility guidelines

The following chapter will describe the design of the TinEye tool, a simple console application that will process images and videos highlighting compliant and not compliant text according to a set of guidelines. The guidelines checked by TinEye will be the ones that ensure text has a minimum size relative to screen resolution and that it has sufficient contrast against its background to be readable by color vision deficient people. This chapter will outline the features it will offer, and the different configuration possibilities for the user.

3.1. Motivation

The usual and straightforward way to ensure that games comply with a minimum accessibility standard is through tedious manual testing. The following processes are the ones currently used by Electronic Arts to ensure that all games they publish are accessible for all players. While teams are aware of the requirements, the validation process is not coupled with the game development. A separate team of testers is in charge of checking that builds are compliant. This workflow checks that in-game text is compliant with internal guidelines based on the ones presented in section 2.3.

3.1.1. Process for text size measurement

Text must have a minimum height relative to the screen resolution it is in. For example, for a 1080 pixel tall screen text must be at least 26 pixels tall (Microsoft, 2021). The manual tester workflow runs as follows:

1. Take screenshot of areas of the game where text is displayed.
2. Open screenshot in image editing software like MS Paint.
3. Measure the distance from the top-most pixel of the word to the bottom-most (figure 3.1).
4. Create pass or fail report depending on obtained measurements.

This process introduces human-based inaccuracy when measuring, since if there are not ascender characters ('h' or a 't') and descender characters ('p' or 'q') present, the tester

can either report a smaller size or make up how tall they expect the full-sized characters to be.

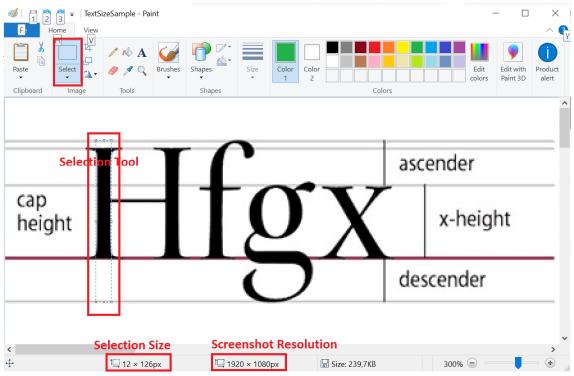


Figure 3.1: Text size measurement in MS Paint

3.1.2. Process for text contrast measurement

On the contrast side of visual accessibility guidelines, the text color must have a minimum contrast ratio of 4.5 against its background so that it can be read correctly regardless of the color combination chosen (W3C, 2008a). The procedure for checking contrast runs as follows:

1. Take a screenshot of the area of the game with text present.
2. Open screenshot in image editing software.
3. Use the eyedropper tool to obtain the hex code pertaining to the colors of the text and its background.
4. Input the color values into a contrast checking tool such as the one in figure 3.2. This website is the one currently being used by quality assurance teams that have to manually check text contrast when testing games.
5. Generate a pass or fail report depending on tool's output.

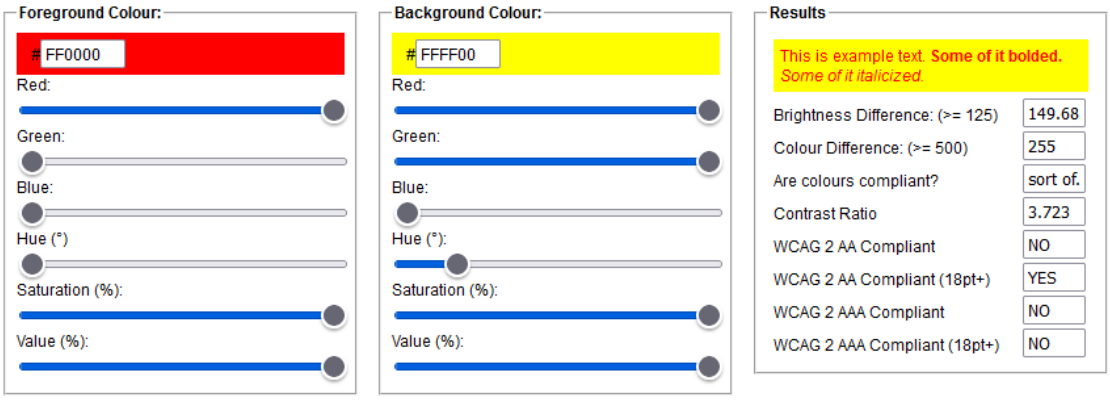


Figure 3.2: snook.ca's Colour Contrast Check tool showing comparison between red and yellow. (Snook.ca, 2015)

3.2. TinEye

TinEye is a tool that automates the described process. The tool will work with the same input as human testers: rendered frames from a video game. It outputs an image with the text boxes overlaid on the frames with their color indicating pass or fail for humans to easily check the correctness of the output. This approach has several advantages:

- The adoption of the tool will be easier in a production environment. Since it is designed as an improvement on the current publishing workflow, not as an alternative one.
- The tool will not be limited to video games. Any image or video with text can be analyzed. This enables validation of text in other settings like GUI applications, cutscenes and movies where text is present like subtitles.
- The tool has no extra cost and presents no overhead to video game development teams in relation to the current process. All the accessibility checks will remain responsibility of the quality assurance teams.

TinEye is designed according to the following pipeline:

1. Loading of the media (images or videos) to be processed.
2. Processing of the media:
   a) Detecting text areas in the media (subsection 3.2.1).
   b) Running font size checks (subsubsection 3.2.2.1).
   c) Running contrast checks (subsubsection 3.2.2.2).
3. Generating the desired report: A log outlining the results for each text box and an image that highlights the compliance status of the text. This image has red boxes around non-passing regions and green ones around passing ones (subsection 3.2.3).

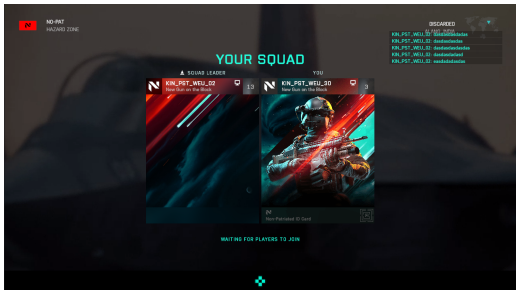
The following subsections will describe in detail the main steps of the pipeline.

3.2.1. Text detection

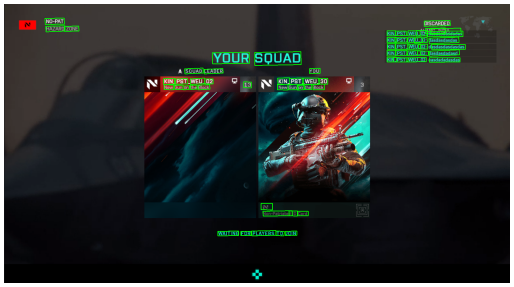
This stage is the most complex of the tool. It is in charge of replacing a human's ability to detect text in order to run the necessary checks on it.

It was decided that text detection would precede both checks. This would help separate the text detection, necessary for both the contrast and size checks. The text recognition is only required exclusively for the font size aspect for getting the number of characters to obtain an average character width.

The detection stage will receive the media and output a list of rectangles where text has been found as seen in figure 3.3b. The cited figure is not a part of the final output of the tool, but is useful in order to understand the process of the tool and see what text is actually being detected and analyzed.



(a) Menu from videogame Battlefield 2042



(b) Detected text boxes in Figure 3.3a

Figure 3.3: Comparison of a game image to its version with text detection

3.2.2. Guideline Checks

After obtaining the list of text rectangles the checks for font size and contrast are run.

The user can configure a set of minimum values that the text has to comply with regarding size (the minimum pixel height in relation to the screen resolution) and contrast (the minimum contrast ratio the text has to have with its background). The tool will execute the necessary calculations to ensure that the text complies with the thresholds set by the user.

3.2.2.1. Font size checks

This stage is responsible for measuring each individual rectangle of text found in the text detection, making sure its size matches or surpasses the one specified in the configuration, and outputting whether each text instance passes or fails. The process followed will be:

1. Obtain a binary mask for the text. The detected text boxes can include some padding so through threshold operations we can get a mask that follows the exact shape of the text to get a more accurate measurement of the text dimensions.

2. Find the position of the pixels belonging to the rightmost, bottommost, leftmost and topmost edges of the text to calculate its size in the next step. When referring to these positions the terms X_{max}, Y_{max}, X_{min}, and Y_{min} will be used respectively.

3. Calculate the height of the text: $Y_{max} - Y_{min}$.

4. If required, run text recognition to identify the word and divide number of characters by text width ($X_{max} - X_{min}$) to calculate average character width.

5. Compare the results with the guidelines and return a pass or fail result for each text box.

3.2.2.2. Text contrast checks

Using the text boxes from the text detection, this stage calculates the contrast between text and its surrounding background for every text box detected. It then outputs whether each box passes or fails.

The process applied for calculating contrast is the following.

1. Obtain a binary mask for the text. Applying the same threshold operation from the font size check we can separate the text from the background, one region of the mask determines the text and everything else that is not inside that mask is the background.

2. Use the text mask to get another mask for the text outline. This mask will be used to calculate the luminance of the background that immediately surrounds the text instead of the background of the whole text box.

3. Calculate the average luminance for both the text and its background using the masks.

4. Obtain contrast ratio between luminance values.

5. Compare the results with the guidelines and return a pass or fail result for each text box.

3.2.3. Report generation

Once the analysis is finished, the tool generates two output images. One for the size tests and one for the contrast tests. Each one will be identical to the original, but will have boxes over detected text that will vary in color depending on the text result. Compliant text will be highlighted in green color, while failing text will be highlighted in red color. If the results for a piece of text are not clear enough to provide a pass/fail status, like not being able to count the characters in a word to calculate width, the text will be highlighted in blue. Yellow, a common color for warnings, is reserved for future expansions of the tool in which an user can have a required minimum size and a best-to-have minimum, text boxes that fall between those two values would get a warning.

3.3. Configuration

The tool has a flexible configuration for values that developers and users might be interested in. From the viewpoint of a user, the most interesting configuration values are the guidelines and the focus and ignore regions. The guidelines configurations set the minimum values the different tests must check for to ensure accessibility, while the regions of interest affect how the tool processes the media:

- **Guideline configuration** sets the minimum standards to adhere to:
  - The contrast ratio between relative luminance text must have with its background.
  - Radius for the text outline that is considered to be the text's background.
  - A list of resolutions with their specified sizes for minimum text height and minimum average character width.
- **App settings** changes the way the tool executes, so the user can have some control over what they want the tool to do.
  - Focus regions: Rectangles that define regions in which to focus the analysis of the image and ignore everything outside.

- Ignore regions: Rectangles that specify regions to be ignored, even if it is inside a focus region.
- Use text recognition: Turn on or off text recognition. Turning off text recognition speeds up analysis but width cannot be measured.
- Print values on result: Whether the user wants the specific measurements obtained in each check to be present in the output image. This way the user can easily see, if text is failing, by how much they would need to increase contrast or size to make it pass.

3.4. Conclusion

This chapter has explored accessibility guidelines and the manual process followed to validate them. Automation is direly needed in this field. TinEye is a proposal that automatically checks this guidelines in a way that is both more exhaustive and precise than manual testing. This proposal is promising and flexible, it can be applied to various other settings other than only video game user interface validation by changing its configuration.

Chapter 4

Tool Development

This chapter describes the development of TinEye for checking if a piece of given media, specifically screenshots or videos from video games, is compliant with a set of guidelines that ensure text is readable to users with deficient color vision or myopia.

The whole development process will be explained, starting out with a prototype phase to make sure our approach was viable (section 4.1), followed by the proper development of the tool (section 4.2) and its subsequent iterations.

Afterwards, its final architecture will be explained (section 4.3), as well as the possible configurable variables for developers in section 4.3.3. Subsequently, the methodology and techniques used during development (section 4.4) will be described. Finally, the various tests that were designed to ensure the tool works as intended will be detailed in section 4.5.

4.1. Initial prototypes

The development for the TinEye tool started out with a series of prototypes. These prototypes helped the team to familiarize themselves with the chosen technologies as well as solving individually the partial problems the tool will tackle. The main objectives for the prototypes, aside from being a first contact with the libraries, were the following:

- Precisely calculating contrast and perceived luminance (subsection 4.1.1).
- Testing the text recognition libraries for measurement (subsection 4.1.2).
- Seeing how EAST would behave with videogame footage consisting of a scene and a HUD (subsection 4.1.3).

During development, the team followed the Agile methodology principles. This was due to the team already having experience working with agile methodologies and that these practices are better when working in small multi-disciplinar teams. These principles are the following:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Additionally, small 2-week iterations (*sprints*) were planned, during which the highest priority tasks were carried out. At the end of each sprint, the state of the tool would be assessed and the priorities for the following sprint would be determined. In line with the second Agile principle, a working version of the tool would always be available to use.

4.1.1. Relative luminance prototype

The goal of the first prototype was calculating the relative luminance ratio between text and its background. This prototype computed the luminance differences between two colors in an image. The result of this calculations can be compared to the minimum contrast value expected by the WCAG guidelines (subsubsection 2.3.2.1) to check if it provides sufficient contrast even for people with some form of color vision deficiency.

Given an image and two pixel coordinates, it had to calculate the relative luminance Y and contrast value between the colors of the two pixels. Its functionality must mimic the 'Contrast Ratio' tool from section 3.1.2.

Using the red, blue, and green components of the colors of an image, relative luminance Y can be calculated with equation 4.1.

$$Y = 0.2126 * R_{lin} + 0.7152 * G_{lin} + = 0.0722 * B_{lin} \tag{4.1}$$

Knowing the relative luminance Y of the colors of both the text and the text's background, the contrast ratio between the two can be calculated with equation 2.1.

Notice that in equation 4.1, color values for red, green and blue are stated as *lin*. This stands for linearized values, which are not the same values you would get when simply opening an image in an image editing software. As mentioned in the explanation of color spaces in section 2.4, colors in these color spaces such as sRGB must be linearized before the relative luminance can be calculated. The formula for linearizing sRGB or ITU-R BT.709 color spaces is equation 4.2 or equation 4.3.

$$R_{lin} = R'^{2.2} G_{lin} = G'^{2.2} B_{lin} = B'^{2.2} \tag{4.2}$$

Formula 4.2 prioritizes calculation speed over accuracy. A more accurate version to transform from sRGB to linearized components is equation 4.3

$$\begin{aligned} &if R <= 0.04045\, then\, R_{lin} = R/12.92\, else\, R_{lin} = ((R + 0.055)/1.055)^{2.4} \\ &if G <= 0.04045\, then\, G_{lin} = G/12.92\, else\, G_{lin} = ((G + 0.055)/1.055)^{2.4} \\ &if B <= 0.04045\, then\, B_{lin} = B/12.92\, else\, B_{lin} = ((B + 0.055)/1.055)^{2.4} \end{aligned} \tag{4.3}$$

Once all components have been linearized the relative luminance Y can be calculated following equation 4.1

4.1.1.1. Results of the relative luminance prototype

A series of tests were ran with small, simple images. These images consisted of a grid of known colors whose contrast could be easily calculated with another tool to check for correctness.

Figure 4.1: Results of comparing a red pixel with a yellow one with the contrast prototype

The final results, shown in Figure 4.1 differ from the referenced website due to different decimal accuracy between programming languages and a different value being used in the linearization calculation. This is because, in equation 4.3 Snook.ca (2015) uses a value of 0.03928 and the prototype uses 0.04045, following the IEC standard (MozillaCorporation, 2022).

4.1.2. Tesseract OCR prototype

The objective of this prototype is to test character recognition and measurement in different scenarios using the Tesseract OCR engine. The measurement should not differ more than 2 pixels from what could be measured by a human carefully counting the pixels using software that provides a grid. This margin of error is acceptable to avoid errors due to anti-aliasing and image compression techniques. The effect of anti-aliasing can be seen in figure 4.2, there appear to be some colored pixels offset from the main character's shape.

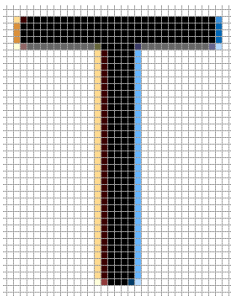
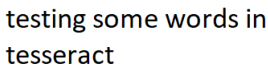


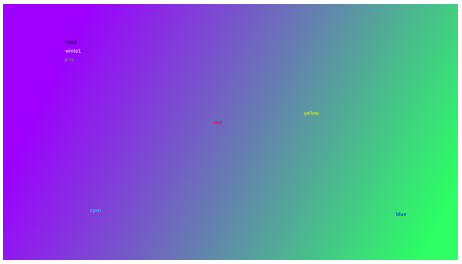
Figure 4.2: Blue and red outline when rendering text with MS Paint

4.1.2.1. Results of the Tesseract prototype

After implementing basic character recognition with Google's Tesseract, the sample images seen in figure 4.3 were processed. Figure 4.3a represents black text on white background, Tesseract's preferred format as a standard OCR. Figure 4.3b presents colored text on a colored gradient, this is outside of Tesseract's preferred input, but as stated by Smith



(a) Black text on white background



(b) Words with low contrast on a gradient background

(c) Luminance map of figure

Figure 4.3: Different media used for Tesseract.

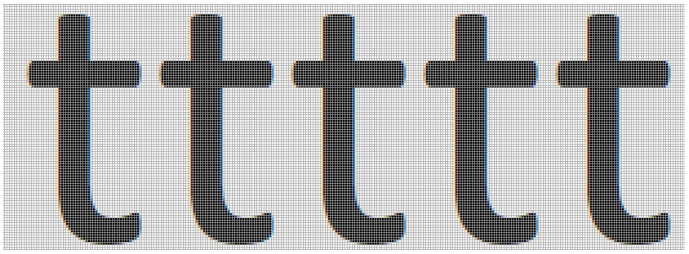
(2007) the OCR is based on patterns and contrast; making the color of the text much less relevant than in other algorithms. Finally, in figure 4.3c is the luminance map calculated with the formula used for luminance in 4.1.2. This would be our preferred input over color images since it takes into account the perceived luminance regardless of the original color.

The results were the following:

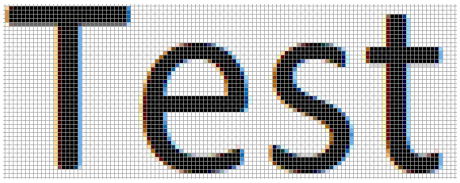
- Figure 4.3a was recognized perfectly
- Figure 4.3b most words unrecognized, only *"cyan"* was recognised.
- Figure 4.3c (Figure 4.3b's luminance map) did not improve results, but in this case only the word *"yellow"* was recognised.

In order to check the size of the text according to the CVAA guidelines (FCC, 2010), we need to get an accurate character measurements. Since Tesseract allows for per-symbol iteration, some images were created and manually measured in order to check the accuracy of the OCR bounding-boxes for each individual character. The created images (Figure 4.4) were all black text over a white background, the objective was to check precision in optimal conditions. The results of the OCR on the presented photos was the following:

(a) letter t, 10x19 pixels



(b) 5 letters t, 47x94px each



(c) test word, characters measuring 31x40 27x32 21x32 19x39 pixels

Figure 4.4: Different media used for Tesseract.

- Figure 4.4a measurements were correct, reporting 10 pixels width and 19 pixels height.
- Figure 4.4b measurements continued to be accurate when concatenating various characters, reporting 5 characters 47 pixels wide and 94 pixels tall.
- Figure 4.4c accuracy problems when testing with real words. The first character from this figure correctly reports 31x40 dimensions. The measurements of following characters are correct height-wise, but the width greatly differs resulting in 51 pixels of width for $/''e''$ instead of the correct 27 or 43 for the last $''t''$ instead of 19.

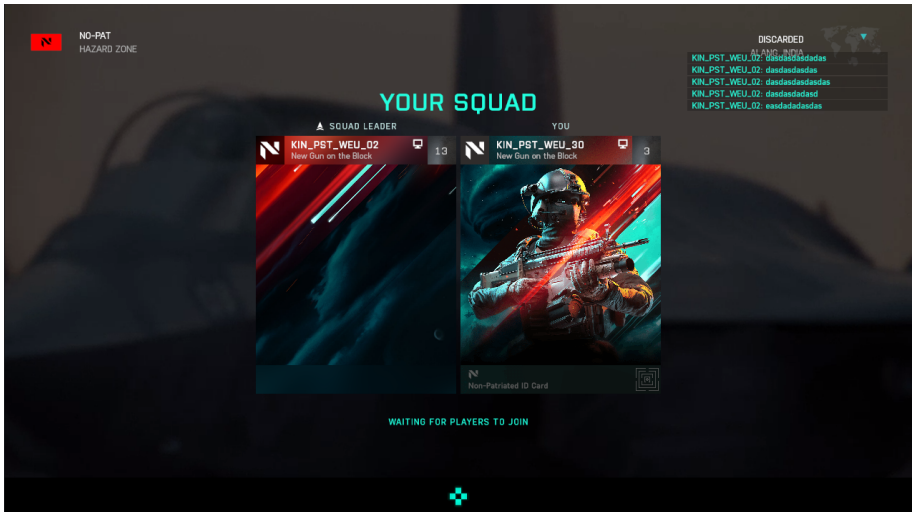
The main takeaways from the presented results were that alternative OCR engines would need to be explored. Tesseract is a state of the art OCR engine but is heavily focused on scanned documents. After some more research, and finding the results on low-contrast discouraging, the discrepancies between the actual accuracy and what was expected from the overview by Smith (2007) propelled further research, finding out that Tesseract dropped support for white text over black background on version 4. This finding confirmed Tesseract's increasing bias towards text documents and raised the need for an alternative OCR solution that could recognize text in more complex scenes and digitally created images with different backgrounds.

4.1.3. EAST prototype for text detection

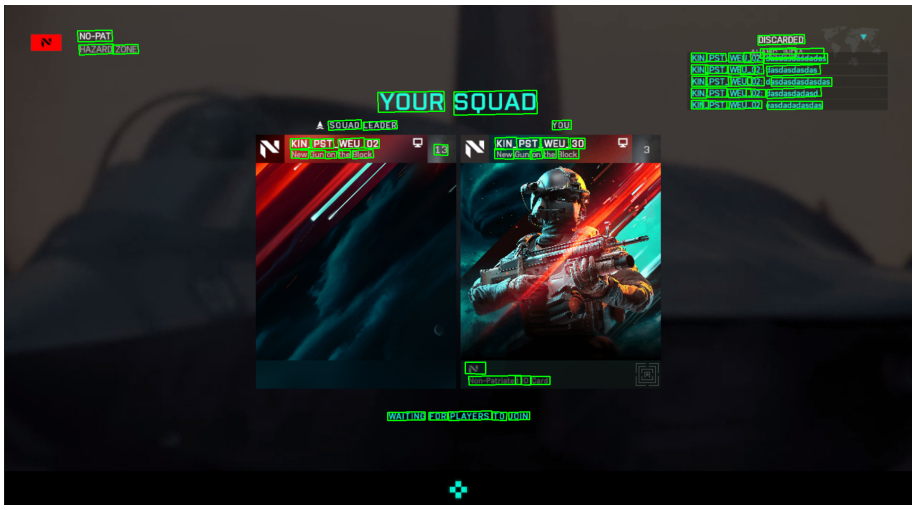
The objective of the following prototype is to see how the EAST algorithm behaves with video game footage. The prototype will involve following a simple example provided in the EAST documentation (Community, 2020) and using it to detect text in a video game's user interface to see if all the text is detected.

4.1.3.1. Results of the EAST prototype

EAST was tested by creating a simple command line tool based on the proposed usage of the sample implementation provided in the EAST documentation and relying in the provided model in Zhou et al. (2018). The results were very promising, showing full recognition of all the text in scene in the test cases (Figure 4.5).



(a) Base Image



(b) EAST detection results

Figure 4.5: Example input and output of the EAST CLI

4.2. Iterative development process

After finalizing the prototypes, the iterative development of TinEyeLib started. The process started with the basic functionalities tested in the prototypes (section 4.1). Once the stages were integrated, a simple client TinEyeApp that loads media and outputs a report using the data from the library was developed.

4.2.1. Iteration 1: TinEyeLib

The first stage of the development consisted in making a library that combined the partial solutions developed during the prototyping process. This library was called `TinEyeLib`, it exposed the necessary methods to test images for text size and contrast. The combination of all three stages created the necessity of having data, like the image's luminance maps or the results that were generated, persist between each stage. A class that held the media being processed and its analysis information was created. This was the `Media` class, the details of TinEye's architecture are explained later in section 4.3.

Once loaded, the media goes through a text detection stage (subsection 3.2.1) that relies on the EAST algorithm. The detected text boxes are saved into the Media class after simple processing. EAST is made to detect text in real world scenes, so the text boxes generated are not always aligned with the vertical and horizontal axis. TinEye's text detection stage takes the result from EAST and creates a rectangle that is axis-aligned to help the text measurement tasks.

After detecting where text is present, the guidelines can be checked independently. The text measurement stage receives a Media reference and iterates through all of its text boxes. During this first iteration, the measurement of the height corresponds with the height of text box, and the width of individual characters is calculated by dividing the width of the text box by the number of characters detected by the character recognition library.

At this point, we realized that we were not content enough with Tesseract's performance. We had hoped that it would recognize in-game text better. This prompted changing the text recognition system to the one provided by OpenCV.

4.2.1.1. Changes to text recognition

As shown in subsection 4.1.2, the results of Tesseract were discouraging. While being one of the most promising tools regarding text recognition in images, Tesseract fell short when dealing with text with complicated backgrounds. The prototype could not recognize text on top of a gradient, even in conditions with good contrast (Figure 4.3b). It is safe to assume that this problems would translate when dealing with more complex backgrounds like those in video games or text that presents a low contrast with its background. Another factor against Tesseract is that since it is used to scan documents it attempts to recognize real dictionary words and not just strings of characters, which when attempting to recognize usernames or other user created texts, real words may not be found.

After deliberation, it was decided that Tesseract's capabilities would not be sufficient to ensure the correct functioning of the tool. OpenCV bundles text recognition capabilities, which provide neural network support for various types of character recognition. The decision during development was to keep the text measurement stage flexible in order to compare the results for both OpenCV and Tesseract to finally choose one or the other depending on their performance.

When implementing OpenCV character recognition inside the text measurement stage, the results vastly improved on those of the Tesseract prototype as can be seen in figure 4.6. OpenCV's solution recognizes much more text, most notably when diverse backgrounds are present. Transition to use OpenCV's provided OCR technology from Tesseract was a success, and now the luminance calculation part of the tool could be tackled.

(a) Detected text regions in a screenshot of a game.

```
[info] Word detected: 6761223
[info] Word detected: testing
[info] Word detected: catchni267
[info] Word detected: testing
[info] Word detected: [VOIP]
[info] Word detected: Dummy
[info] Word detected: 786761223
[info] Word detected: Dev
[info] Word detected: ChWEPOstDOOG
[info] Word detected: 924318983
[info] Word detected: MENU
[info] Word detected: MAIN
[info] Word detected: 2Dev
[info] Word detected: CatchwEPostd
```

(b) Results of running image Figure 4.6a through OpenCV's OCR algorithm

```
[info] Tesseract word detected: catchnl1267
[info] Tesseract word detected: ┬¼
[info] Tesseract word detected: 6761223
[info] Tesseract word detected: ┬¼
[info] Tesseract word detected: %
[info] Tesseract word detected: -
[info] Tesseract word detected: S
[info] Tesseract word detected: ┬¼
[info] Tesseract word detected: 2
[info] Tesseract word detected: Dev
[info] Tesseract word detected: 924318983
[info] Tesseract word detected: ┬¼
[info] Tesseract word detected: ┬⌐
[info] Tesseract word detected: MAIN
[info] Tesseract word detected: MENU
```

(c) Results of running image Figure 4.6a through Tesseract with a minimum confidence of 50%

Figure 4.6: Performance comparison of Tesseract vs OpenCV's text recognition

4.2.1.2. Text background calculation approach

The remaining stage would be that in charge of calculating the contrast of each text piece with its background. This stage needs both a way of calculating relative luminance and a way of separating text from its background. The contrast calculation was handled in section 4.1.1. To scale the described prototype, the luminance values of the whole image are calculated at once taking advantage of OpenCV's parallelization when applying a function to all pixels in a matrix.

The method for separating text and its background at this point is a naive simplification. The mean luminance of each text box is considered to be the mean luminance of the text, and the outer edge's mean luminance of the text box, the background luminance, as can be seen in figure 4.7. Once the contrast is calculated, the comparison result against the guideline threshold is stored in the Media results.

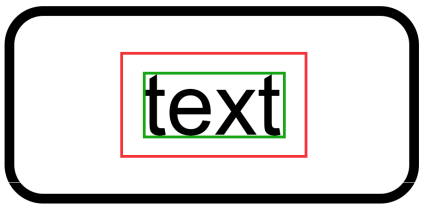


Figure 4.7: Example of a naive luminance calculation.

4.2.2. Iteration 2: Increasing accuracy

After having a first working version of the tool, several improvements were made. Some of these improvements are aimed at improving the accuracy of the calculations. Other improvements such as merging text boxes or ignoring tilted text were aimed towards providing a clearer report.

4.2.2.1. Histogram approach

After completing a first version of the contrast checking stage, improvements needed to be made in order to represent the contrast between the text and its background. The problem with the naive approach described in section 4.2.1 is that it takes into account the immediately adjacent background to the text into the mean of the text itself. This means that when working with thin fonts, the luminance value from the text itself is much less significant than its immediate background.

The first alternative was using histograms. Histograms are the representation of data points grouped into what are usually called bins, whose height is determined by their absolute or relative incidence. They can be used to obtain the difference in luminance between the foreground and the background. Their most representative values in each of their histograms could be used to represent their representative luminance and calculate the relative luminance ratio.

In photography and image processing histograms are usually used to visualize the intensity of the different colors in an image; for color images using three-channeled histograms and for black and white just a single channel. In the case of images with few values for intensity (256 in most consumer cases) the bins used encompass only one value, but for higher color depth images bigger bins may be used. An example of the histogram of a triple channel image can be seen in figure 4.8.

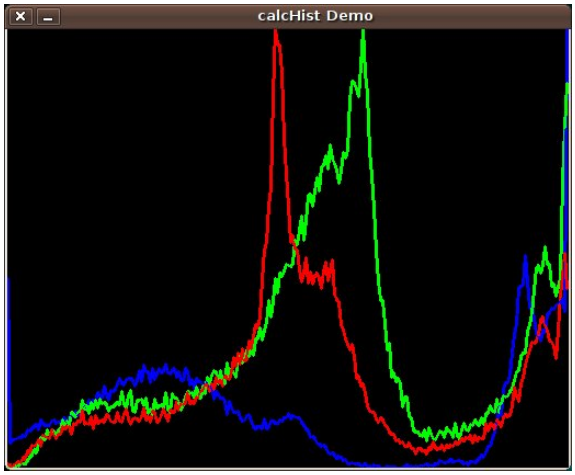


Figure 4.8: Histogram showing the distribution of intensities for each color channel in an RGB image

The proposed approach was to calculate the luminance histogram for the text region and the histogram for the area immediately surrounding the text (representing the background luminance) and subtract one from the other to know the representative luminance for both the text and the background and calculate the ratio. An explanatory diagram of this process can be found in figure 4.9.

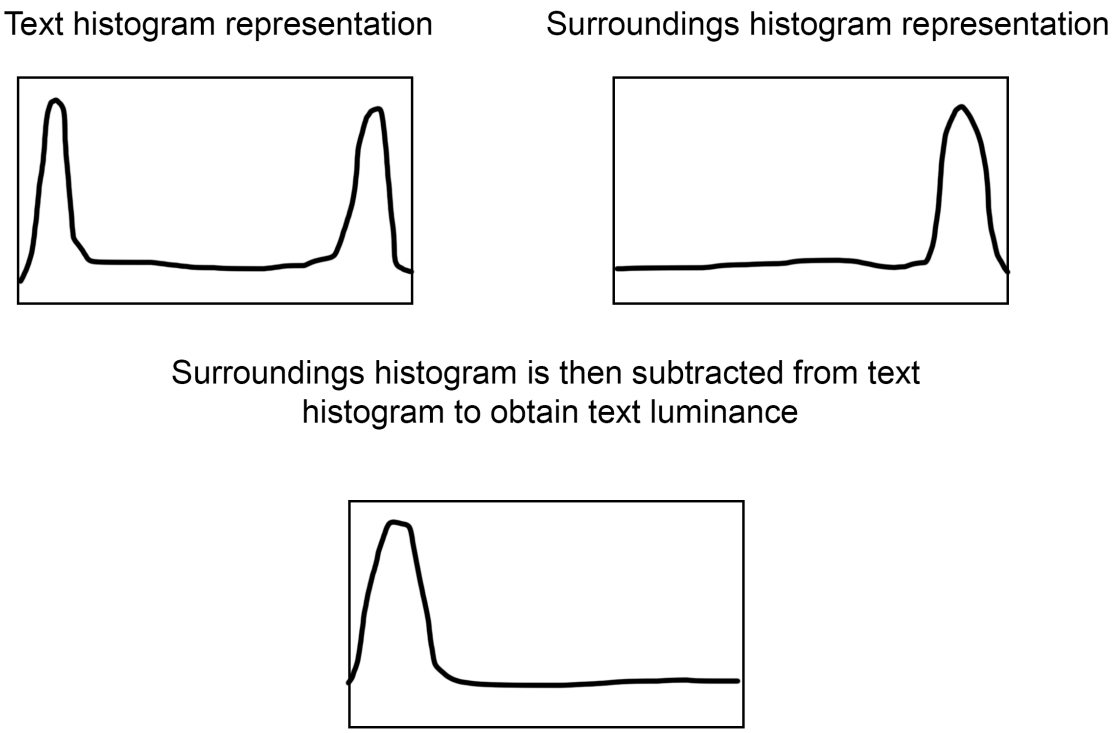


Figure 4.9: Diagram explaining the approach of subtracting two histograms to get the representative foreground luminance

When subtracting one histogram from the other it was found that in certain cases the text's luminance was not representative enough or an erroneous value was being output.

Figure 4.10: Top: Luminance map of a region. Bottom: Mask obtained from applying threshold to luminance map

This could be due to various factors: The text being too thin, making its area compared to the background very small even though it is readable; anti-aliasing 'blurring' the text, spreading out its luminance; or the surrounding luminance being very dark but the area just behind the text being brighter, so when subtracting histograms only part of the 'real background' was being subtracted.

Histogram comparisons ended up not being a viable avenue to calculate luminance ratio between text and background.

4.2.2.2. Thresholding approach

After testing out histograms, it was decided to try a thresholding-based approach. This approach works by separating the pixels corresponding to the text from the ones corresponding to the background and averaging their respective luminance values to later get the ratio. Using masks is an efficient way of separating the text from the background to calculate each one's relative luminance. To create this mask a threshold is applied to the luminance. It automatically separates the darker and lighter regions into a binary mask (figure 4.10).

To apply a threshold is to decide a base value and alter an image depending on each pixel's relation to this value. For example, in a binarization threshold operation a threshold value is set and every pixel in the image that falls below the threshold is set to 0 and every pixel over the threshold is set to 1. This creates a mask for regions with higher luminance, allowing the calculation of the average luminance inside this mask, the average outside, and the ratio between them.

To decide the threshold value the Otsu algorithm (Yousefi, 2015) is applied, which calculates the optimal threshold value based on the present luminance in the region. This value is calculated dynamically depending on the color values of the given region and not set previously because the tool has to process a lot of different text with varying background colors in the same scene.

To obtain a representative value for the background luminance an outline for the text is calculated. The text mask is dilated a certain amount specified in the configuration and then the original mask is subtracted from the dilated one to create an outline mask. The luminance values inside this outline mask are averaged to obtain the background luminance value for that text.

Contrary to text measurement, there is no manual equivalent to this process since the previous process involved a tester choosing two pixels they thought represented the text and its background (subsection 3.1.1). Even without a baseline to compare to, the results

of the algorithm shown in figure 4.11 are satisfactory. This process is more accurate and less reliant on tester behaviour and would be used for TinEye.

(a) Luminance map of environmental text detected in a scene

(b) Text mask of the text detected

(c) Outline of the text mask obtained

Figure 4.11: Process of mask and outline calculation

4.2.3. Iteration 3: Filtering empty text boxes

While working with the EAST algorithm a problem was encountered, sometimes boxes that didn't contain any text or that contained text belonging to the game environment were detected. The algorithm is made to detect text on a real world scene, so sometimes elements of the background or environmental text are detected and sent to the tool (examples in figure 4.12).

(a) Environmental details recognized as text

(b) Environmental text detected

Figure 4.12: Non-desirable text boxes detected by EAST

The first step towards filtering this kind of false positives was to make sure that every detected text box was part of the game's user interface (UI). The most notable restriction was that UI text is usually axis-aligned, the text is not tilted. This realisation prompted the addition of a configuration parameter that would discard text boxes tilted over a certain threshold, reducing the number of false positives consisting of environmental text. The implementation was simple: after getting the unprocessed results from OpenCV, the angle

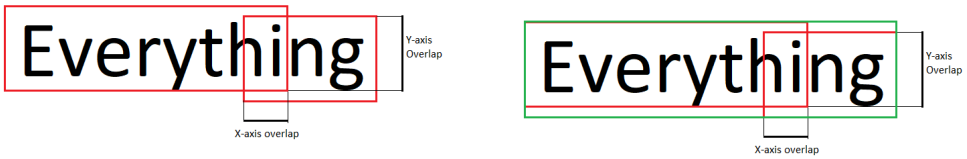
of the text box horizontal lines are checked before deciding if that text box should be sent to the next stage.

4.2.4. Iteration 4: Axis Aligned text box merging

The following post processing step was needed because the EAST algorithm sometimes split up single words. During execution, EAST attempts to recognize everything that is text in a given image and fit it into boxes. For a given word it might detect more than one instance of a text box since it has no notion of what a word is, it just detects text. If two boxes are sufficiently overlapping eachother it must mean they are part of the same instance of text, so it merges them. To merge these overlapping textboxes EAST uses NMS (non-max suppression), but in some cases this process proved not to be enough for our tool's needs as some overlapping text boxes were left over.

The NMS algorithm takes into account the intersection area between to overlapping rectangles that outline the same figure and their confidence to make sure there are no duplicate results (Sambasivarao, 2019). Since we are working with axis-aligned text we have an advantage over a more general solution for overlap detection like NMS, the overlap needed for two rectangles to be determined as being the same occurs mostly only on the horizontal axis.

After instantiating and filtering the text boxes, the box array is looped through and checked for overlap between boxes as shown in figure 4.13a. Different minimum thresholds can be configured for overlap detection on the X or the Y axis. For example, in languages that write horizontally the thresholds would needed to be high for the Y axis but low for the X axis (meaning that most of the words' height must coincide but they only have to overlap a bit on their horizontal axis). If both requirements are fulfilled, both text boxes merge into their union as shown in figure 4.13b



(a) Example of a split text box. Presents high overlap in Y axis, but low on X axis.

(b) Result of a successful text box merge

Figure 4.13: Process of merging two overlapping text boxes

4.3. Architecture

The TinEye project is comprised of three modules:

- **TinEyeLib** is a static library that allows its users to analyze images or videos to see if they comply with text size and contrast requirements and get back pass or fail reports. TinEyeLib implements the core functionality of the tool. It implements methods that allow the fast calculation of luminance maps, detecting text boxes and calculating both the required measurements and the contrast ratio between the text

and its background. The Library exposes both the TinEye class, with all the methods detailed, as well as Media factory to allow the end-user to handle resource loading.

- **TinEyeApp** is a simple console application made to use TinEyeLib and generate a report with the data provided. This application loads the media to be analysed, instantiates a TinEye class with the desired configuration and tells it to process the media. Once the results are given, TinEyeApp will generate two copies of the original media with the resulting text boxes overlaid: one for font size checks and another for contrast checks. These copies will outline compliant text in green and non-compliant text in red.
- **TinEyeLib.Tests** features a suite of tests that covers most of the image manipulation methods. An example of this is making sure the luminance and contrast calculations results match the ones from Snook.ca (2015). In addition to this, there are acceptance tests that check the expected results of a full run of the application. The devised tests are described in detail in section 4.5.

4.3.1. TinEyeLib

The architecture for TinEyeLib is simple, it exposes the Media, Results and TinEye classes allowing for different aplications to be built with it. The diagram presented in figure 4.14 shows what classes are available to the end user from the whole architecture. This API allows the user to

- Creating a TinEye instance with a specific configuration
- Loading Media
- Giving a Media instance to a TinEye instance in order to analyse
- Accessing a Media instance results

4.3.1.1. TinEye

TinEye is the executive class of the library. Once it is initialized with a configuration and load media, it provides the methods necessary to analyse media and run the contrast and size checks.

This class holds its own configuration, as well as a reference to a text detection class instance.

4.3.1.2. Media

Media is an abstract class that provides a factory method to load videos and images. Video and Image are subclasses from this class that implement specific logic.

The class holds a Results struct with the information needed to build the output. Results are an attribute of Media because there is no reason for the results to have a longer memory lifespan than the media they describe. This approach comes in handy when dealing with long videos, since there is no need to keep all the frame results in memory.

The Media class serves as a container for both the image and luminance map of the active frame as well as implementing many methods to save execution results as files when

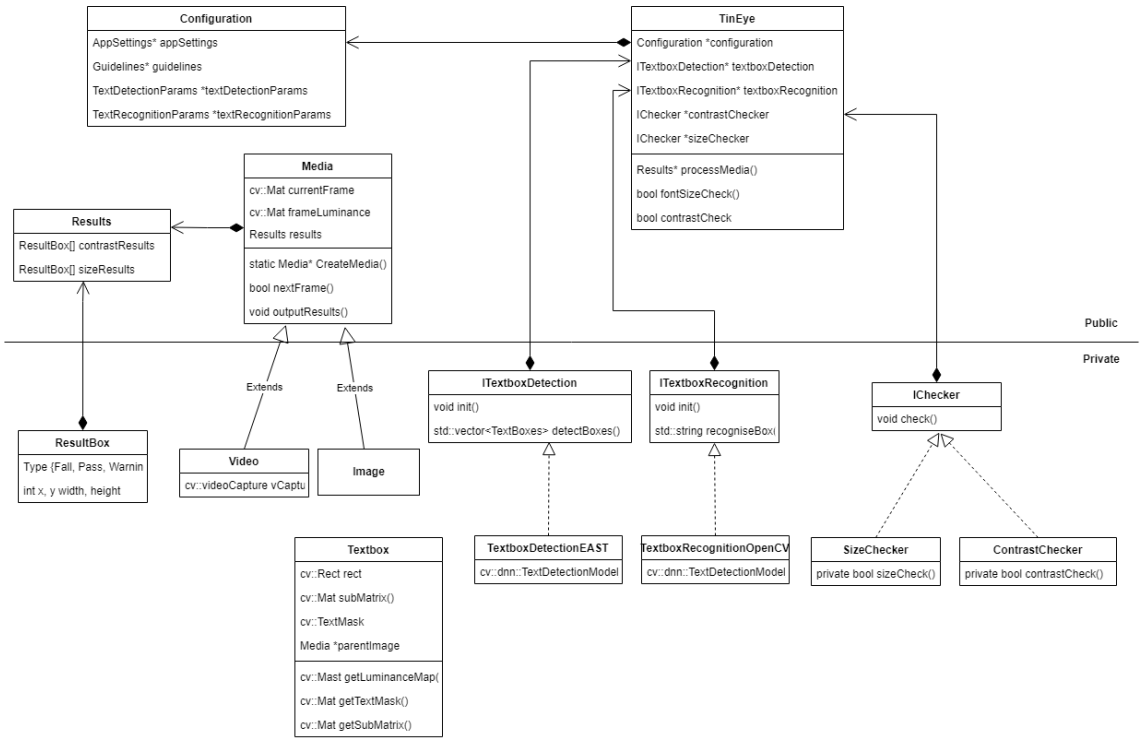


Figure 4.14: TinEyeLib Class Diagram

specified by the configuration: saving the image matrix, luminance map or a submatrix of any of them.

Since media has access to the frame that has been processed and the results it generated, it can create the output images with the size and contrast check results. This way the user can visually understand the results instead of having to parse the result struct by themselves.

4.3.1.3. Results

The Result struct holds an array of ResultBoxes for each type of check and an overall result to avoid iterating through the whole vector if a general result is needed. ResultBoxes are also structs. Each ResultBox has the data regarding the position and size of a text box is and a pass status that can be a PASS, FAIL or a WARNING.

4.3.1.4. Textbox

This class holds the data related to a detected piece of text, mainly its position, width and height, as well as references to the submatrix of the original image they correspond to, its luminance map, and a binary mask that separates the text from the background.

4.3.1.5. ITextboxDetection and TextboxDetetionEAST

The ITextboxDetection interface is an abstraction to detect text in the image and return it as a series of Textboxes. The current implementation uses the EAST algorithm to detect text and processes the results to fit out specific needs of detecting text in UI. This class is constructed following the parameters specified by the configurations and implements methods to output an array of text boxes as well as processing the EAST results. If a

better algorithm is developed, this abstraction would make it easier to implement and compare with the current method.

4.3.1.6. ITextboxRecognition and TextboxRecognitionOpenCV

This interface is in charge of abstracting the detection of text inside a textbox. The current implementation receives a Textbox and uses OpenCV models to recognise it, but more implementations can be developed in the future.

4.3.2. IChecker, ContrastChecker and SizeChecker

This interface allows us to abstract the size and contrast calculations from the main application. The main goals of this abstractions are the ability to provide alternate size and contrast calculations in the future and to be able to add new checks if proven necessary.

4.3.3. Configuration

The tool needed to be flexible and allow for different configurations based on the language and local accessibility laws applied to the media being processed as well as different guidelines the developer might want to adhere to. Minimum required values need to be easily updated, and from the technical side, text detection and processing needs to be easily configurable to adjust accuracy and set of used characters.

The configuration comes in the form of a `.json` file, due to its ease of use and that it is easy to parse with a simple library.

The Configuration class is a simple parser that holds small classes with compartmentalized information for different stages of the tool, shown as its attributes in figure 4.14. This attribute classes are not shown in the diagram for simplicity, since they are simple classes consisting of getters and setters for the following data that should be present in the `json` file:

- **Guideline** sets the minimum standards to adhere to:
  - Floating point value determining the minimum contrast between relative luminance that text must have with its background (section 2.3.2.1).
  - Integer value for the radius of the text's outline that is considered to be the its background (mentioned in 4.2.2.2).
  - A list of resolutions with their specified sizes for minimum text height and minimum average character width as integers.
- **App Settings** configures what information is saved during execution of the tool, so the developer can visualize the intermediate steps taken and debug as needed. It also changes the way the tool executes, so the user can customize it to their liking
  - Use text recognition: Boolean value that turns on or off text recognition. If it is turned off, it speeds up the tool, as it does not attempt to recognize text in a text box, but it cannot measure its width.
  - Save text detection stage results: Boolean value for whether show the unprocessed results of the EAST text detection as seen in figure 3.3b.
  - Save luminance map: Boolean value for whether to output a clone of the media that represents the luminance (section 2.3.2.1) of each individual pixel.

  - Save luminance masks: Boolean value for whether to save the text mask and its outline to memory.
  - Focus regions: Rectangles that define regions in which to focus the analysis of the image and ignore everything outside. They are defined as a list of objects that have x and y coordinates as well as a width and height.
  - Ignore regions: Rectangles that specify regions to be ignored, even if it is inside a focus region. They are defined the same way as focus regions.
  - Print values on result: Boolean value if the user wants the specific measurements obtained in each check to be present in the output image. If the flag is on, when generating the output image, the result value is attempted to put to the right of the text box. If the text would not fit in the image when being put on the right it is put on the left.

- **Text detection Parameters** allows the user to configure EAST parameters and other operations relating to text detection, like how many degrees a text box can be rotated before being marked as environmental text.
  - EAST parameters: Floating point values for non max suppression algorithm, minimum confidence, detection scale and mean.
  - Rotation threshold degree: Integer value expressed in degrees. If a detected text box is rotated more than this threshold value it will be discarded.
  - Merge threshold: Floating point values in a range from 0 to 1 for the x and y threshold for merging. If two text boxes overlap over a certain amount they will be merged into a singular text box (details on how this works in section 4.2.4).
- **Text recognition parameters** that allow for configuration of the text recognition module's parameters
  - Model: The filename of the specified neural network model for OpenCV (4.2.1.1) that will be loaded for recognition.
  - Vocabulary: The filename of the vocabulary file that indicates which characters are expected to be found in scene.
  - Neural network parameters: floating point values for mean, scale, decode type and integer values for input size.
- **Linearization values for sRGB**: A list of 256 precalculated values for the sRGB linearization. This prevents the floating point error that appears when attempting to linearize sRGB values during runtime.

If a children object is malformed or missing from the `json` file, the parser in `Configuration` will create one from fallback values from the source code. If the whole configuration file is missing or malformed, all the member classes will be built from these fallback values.

4.4. Continuous integration

The version control system used for the project was git, due to the teams familiarity with it and it being widely used across the software industry. As an extra step to ensure that the whole team was always familiar with what was going on in the code we instituted mandatory code reviews for every change to the code base. That way we were always up to

date with anything that changed and the code passed through a second set of eyes before being merged into the main branch, preventing possible errors. This approach was possible because of the limited size of the team, consisting of two members.

To ensure TinEye always presented a working version, both in a windows development environment and a linux-based distribution, the Electronic Arts members provided our team with a continuous integration pipeline. Usually, when working on software in teams of various people, when one person wants to introduce a change into the codebase they create a request to enact these changes. Another developer has to come along and check their code, download it, and test it locally to make sure that everything works correctly.

This is where the continuous integration pipeline comes in. When a developer wants to add their changes to the software, an automated build process is executed that checks that everything builds correctly and runs previously designed tests that ensure the program works as intended. After these automatic checks are run another developer can simply confirm the correctness of these checks and merge the changes right away (Martin Fowler, 2006).

The provided pipeline ran on a Debian docker image had 3 functions:

1. Building the C++ project
2. Running a set of tests devised by us
3. Checking for memory leaks, by running a second time our test suite

The project build stage loads all of the necessary dependencies onto the docker image, like OpenCV or the C++ compilers, and compiles all of the projects related to the tool: The main library, the console application and the unit tests.

If the compilation fails on any of these projects the pipeline execution stops and raises an error.

4.5. Tests

A battery of tests, ranging from unit tests testing out specific functions in the code to acceptance tests that make sure that passing test cases always pass and failing ones always fail, were devised to make sure that various aspects of the tool were working correctly. They ensure that the tool would not break when performing incremental advances to the code or when refactoring some section of it.

The tests can be divided into the following categories:

- Luminance calculation checks
  - A white image has a mean luminance of 1.
    A completely white image should have a mean luminance of 1, the maximum luminance value. This test helped us discover that there was an approximation error when calculating the luminance with floating point decimals after linearizing RGB colors. The error was solved by implementing look up tables with precalculated values.
  - Luminance mean calculation with masks.
    Selecting a region of an image with a mask should only calculate the mean for the specified regions, allowing for the background and foreground luminance calculations.

- Maximum contrast.

  The specified maximum relative luminance contrast is 21, the ratio between the luminance values of pure white and pure black. This test ensures that both the luminance extraction of these colors and the contrast between them are correct.

- Contrast ratio is commutative.

  Contrast ratio calculations should be commutative, the result of the ratio calculation should be the same whether the first operand is the lightest value and the second the darkest and vice versa.

- Image operations checks

  - Luminance flip of a complete image.

    When flipping an image's luminance each of the resulting pixels' luminance should be one minus their previous luminance.

  - Luminance flip of a region.

    Its behavior is the same as the previous test but pixels outside of the specified region should remain unchanged.

  - Double luminance flip.

    When flipping an image's luminance twice it should be back to its original state.

  - Double luminance flip of a region.

    As with previous tests, the specified region of an image should be unchanged when flipping its luminance twice.

- Contrast checks

  A series of images were devised with specific color values near the threshold in different circumstances to make sure that the tool responded correctly. High contrast, low contrast, and flat, gradient and striped backgrounds were used. These test images can be seen in figure 4.15

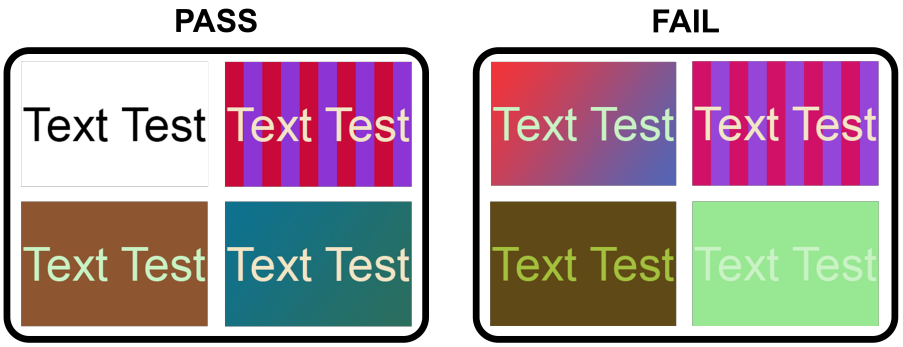


Figure 4.15: Diverse contrast checks to make sure calculations are correct

- Guideline tests

  The tool always tries to load a configuration file on startup. These tests ensure that this configuration adheres to legal guidelines and in case of a malformed or missing configuration there are default fallback values that work correctly.

- Size tests

A series of images were created for failing and passing cases for a variety of fonts for the default resolutions the tool supports (720p, 1080p and 4k). The types of fonts tested include serif, sans-serif, mono-space, and special fonts with wider and thinner characters. An example of various fonts tested can be seen in figure 4.16, the actual tests only feature one type of font at a time.

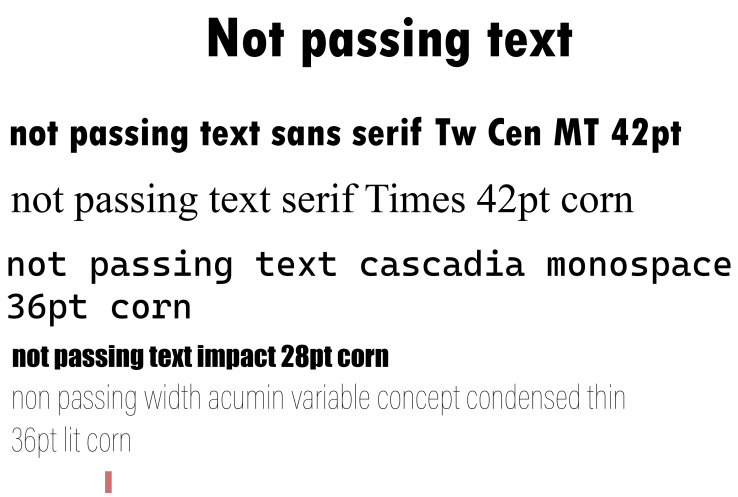


Figure 4.16: Example of various fonts not passing size tests in 4k resolution

4.5.1. Memory leaks and address sanitation tests

After passing all of the previous tests the application runs them all again while monitoring the memory usage of the tool. This makes sure that no dynamic memory has been left unaccounted for when completing execution and that it has been released, as well as making sure that no pointers were left pointing to freed regions of memory during execution.

4.6. Conclusions

In this chapter we have covered the process for developing the TinEye tool. First, some prototypes were made to assess the capabilities of the technologies chosen for the development and the viability of the approach. Another objective of this prototyping was finding out the level of accuracy we could expect from the EAST algorithm, developed for real life scenarios, when working with videogame user interfaces. The remaining objective of the process was to find out a suitable library for recognizing said text.

After the prototyping process was finished, the tool was developed following an iterative process. The team worked in two week iterations that always focused on having a working product of the tool. This allowed the tool to receive constant feedback, both from our tests and from peers at Electronic Arts, whose needs shaped the way the tool behaved. The first iterations of the tool were focused on integrating what we learned during the prototyping process: detecting text with the EAST algorithm, size and contrast checks, and using OpenCV for text recognition when necessary for measurements.

After the tool was sufficiently advanced, a series of real test cases were run through it to check its accuracy. The following iterations focused on improvements to said accuracy, with measures to avoid environmental or false text detection, more accurate text size

measurement and better selection of meaningful background pixels for contrast checks.

TinEye's resulting architecture was described, explaining the different modules the project is split into and how the different classes relate between each other. Finally, the test suite comprised of different unit and acceptance test to ensure the tool's correct operation as the codebase changes was explained in detail.

Chapter 5

Evaluation and discussion

This chapter goes over the profiling of the tool, analyzing which stages of the process take up more execution time. Additionally, the study of various real world test cases that brought up the existence of certain edge cases that bear consideration when attempting to automate the entire process.

Afterwards, the implication of this evaluation, the accuracy of the different analysis the tool does, and the impact and improvements over the current accessibility validation methodology the industry uses will be discussed.

5.1. Profiling

The core metric to TinEye, other than analysis correctness, is execution time, so this would be the focus of the profiling process. The main objective of the profiling process was to see which functions and stages of the program were taking the most time.

The first approach was the use of the integrated profiler in Visual Studio 2022, the IDE of choice for the development. The integrated profiler would allow to see a call graph with information of execution times, and percentages of total runtime taking into account the time taken by a function and its called functions. The results from this tool showed that most of the execution time was spent on external code implemented by OpenCV.

The first run of the Visual Studio profiler proved to be almost too precise for our needs. What we wanted to study was the proportional time taken to execute each step of the analysis process: Media loading, luminance map generation, text detection, text recognition, mask calculation, text measurement, contrast ratio calculation, and result generation.

For this reason we opted for a lightweight header written by Chernikov (2019). It was chosen for its ease of use, simple results to analyze afterward, easy setup and cross-platform compatibility. This simple library uses the class `InstrumentationTimer` to measure the time it takes for a function or scope to be executed. Once the timer stops, the data is written in a json file compatible with Google's Chrome tracing view in `chrome://tracing` that displays a hierarchy showing the execution time of a timer and the ones used while it was running as shown in figure 5.1

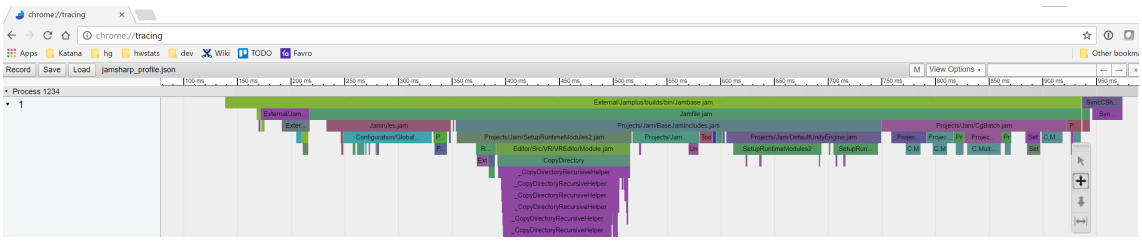


Figure 5.1: Example of the chrome://tracing viewer

Once the profiling tool was chosen, it was determined that the code regions to profile were:

- Initialization: Load the tool's libraries and neural networks into memory.
- Luminance map calculation: Load an image and calculate and store its luminance map information.
- Text box detection: Run the current image through the EAST algorithm, modify and prune the results.
- Size measurement: Text recognition to get number of characters and then obtaining text measurements.
- Contrast calculation: Mask obtaining and calculation of average luminance values inside mask area.

5.1.1. Results and conclusions

After setting up the profiler, the tool was run on release configuration analyzing a set of images with 1080p resolution. The results are shown in figure 5.2.

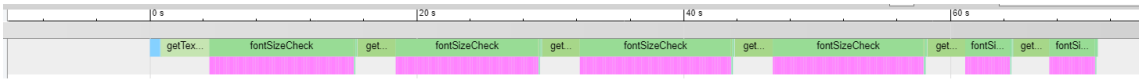


Figure 5.2: Windows release profiling view

The majority of the execution time is spent recognizing text (*fontSizeCheck* function), followed by detecting text boxes with the EAST algorithm (*getTextboxes* function). Execution times for optimizations such as axis aligned text box merging, luminance map calculations, or filtering tilted text boxes are trivial in comparison to the text detection and recognition. The contrast checks, although reliant on the text box detection step are also fast.

To improve the overall performance of TinEye, the following measures were considered:

- Text recognition, while only being used for average character width calculation, takes up 80% of execution time. During the test process, we were unable to find a font that would fail the average character width requirement while meeting the height one.

  This optimization has been implemented as an experimental feature, it can make the tool faster, but it also has a minor downside: the possibility of raising a warning when text was detected but not recognized would be lost. This warning would be helpful dealing with edge cases where contrast is too low or text is too small for the text to be recognized.

- When processing a video, since text has to be on screen for a certain amount of time for a human to read it, a lot of frames contain the same information, moreso at higher framerates. To avoid wasting computation time on practically identical frames, an algorithm could be designed to only analyze the more relevant frames. Proposals for this approach are explained more in-depth in chapter 6.

5.2. Real-world-cases testing

A series of images and videos from games like Apex Legends, Knockout City, Battlefield 2042, NHL, or Need for Speed were provided by various departments at EA. These pieces of media were used to test the tool's performance, to see if it functioned adequately and to find any possible bugs or unintended behavior.

In total, thirty-five images and twenty-six videos of video game footage were analyzed. As will be detailed in the following section, the tool lived up to our expectations of its performance. Known issues, like environmental text detection or detection of environmental non-text features as text, came up, but they were to be expected. These results reinforced the design decisions of allowing the user to set an area of interest, or the capacity of ignoring a certain area of the screen.

In the cases the tool was designed to work in, which is accessibility validation in chat and subtitles text, it worked outstandingly well. It marked exhaustively every word recognizable in a scene and performed size and contrast validation on each of them. Various examples of these behavior can be seen in figure 5.3.

In other situations that were not planned for when specifying requirements, like in game menus or external desktop applications, the tool also performed well (as seen in figure 5.4). It makes sense, since the context of the analysis of these applications is similar to analyzing a piece of footage of a game. This opens up the door for possible extensions of accessibility guidelines, since the technology to check for it is available here.

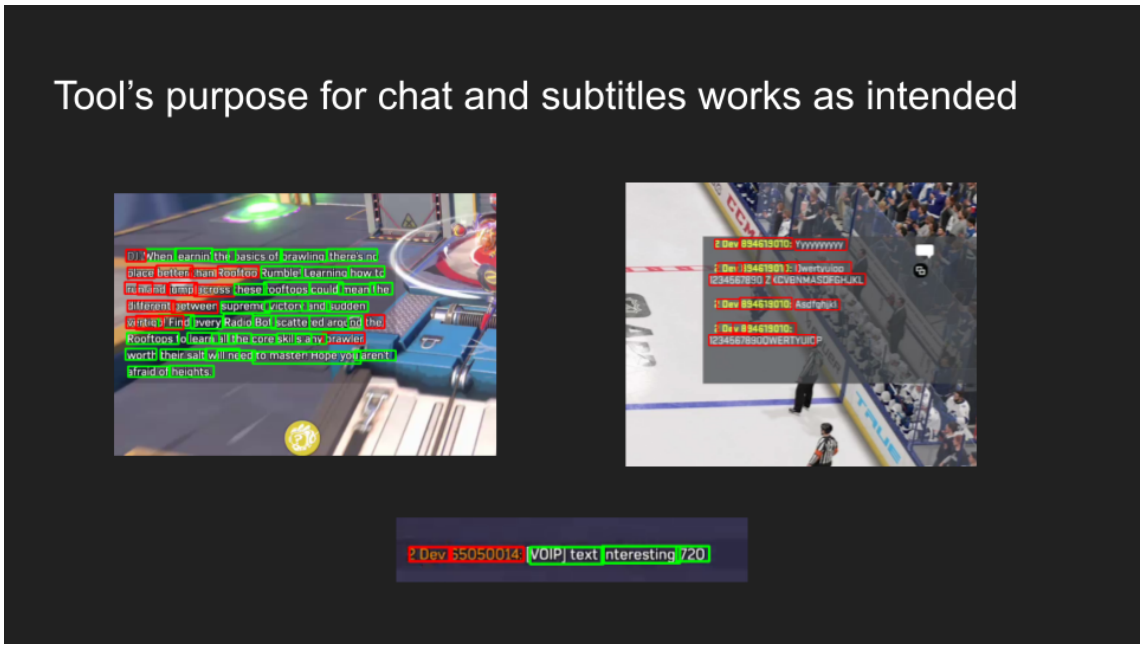


Figure 5.3: Correct cases of image processing, subtitles and chat regions in-game.

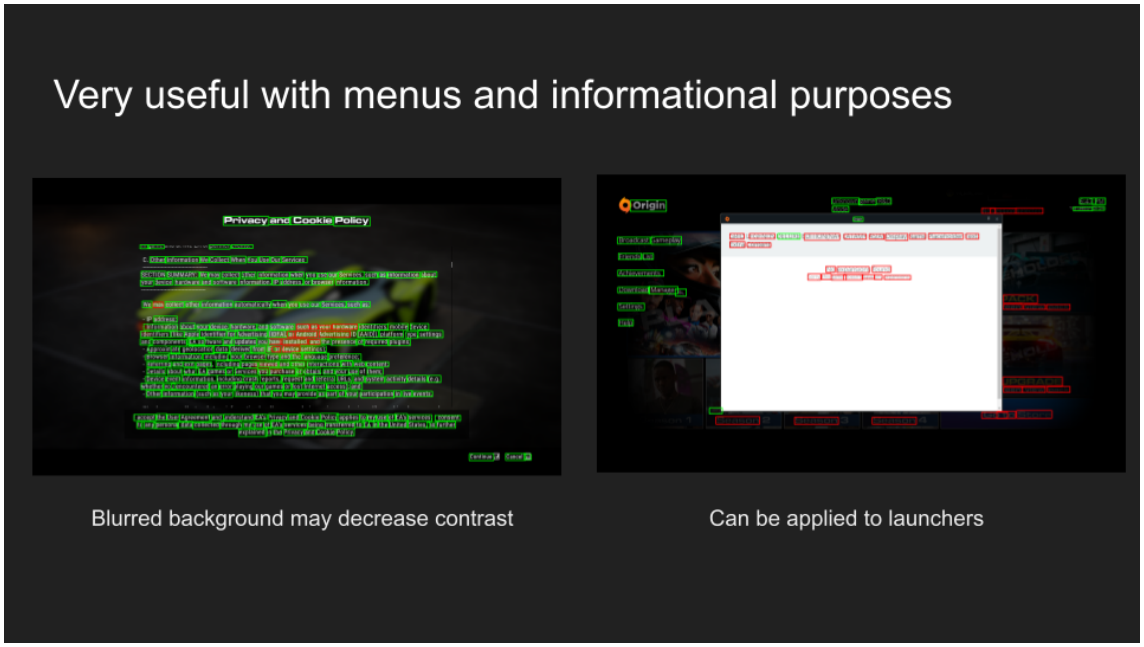


Figure 5.4: TinEye analysis of game menus and desktop applications.

5.2.1. Errors found

As expected there were some situations were the tool's performance was not completely correct. In figure 5.5, certain elements of the game's environment would be detected as text by EAST. Usually, since they are not really text, these instances would lead to a failing results, since they would lack the necessary contrast in their elements or the required size.

Figure 5.5: Environmental details in the background detected as text.

In figure 5.6, text that was present in the environment and was not part of the user interface (like billboards, crates, or graffiti) would end up being processed by the tool.

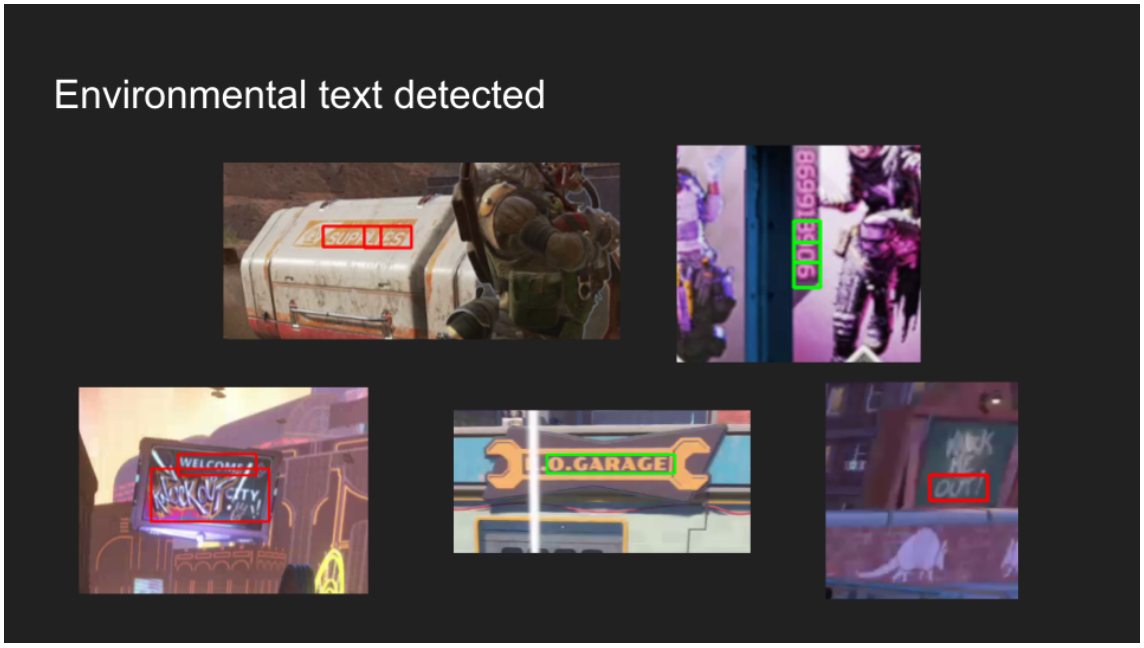


Figure 5.6: Environmental text present in-game detected.

This behavior cannot be really called unwanted, since the tool is recognizing text; which is what it was meant to do. Still, these cases pose a problem since this environmental text can not have enough contrast at times or is really far away and its size does not pass.

Finally, in figure 5.7 certain elements of the user interface that are not text were being detected as such. This issue stems from how the text detection neural network works, it looks at the image and assumes that certain features in it are text and gives a confidence value of how sure it is that that feature is text. Possible fixes to this will be covered later in chapter 6.

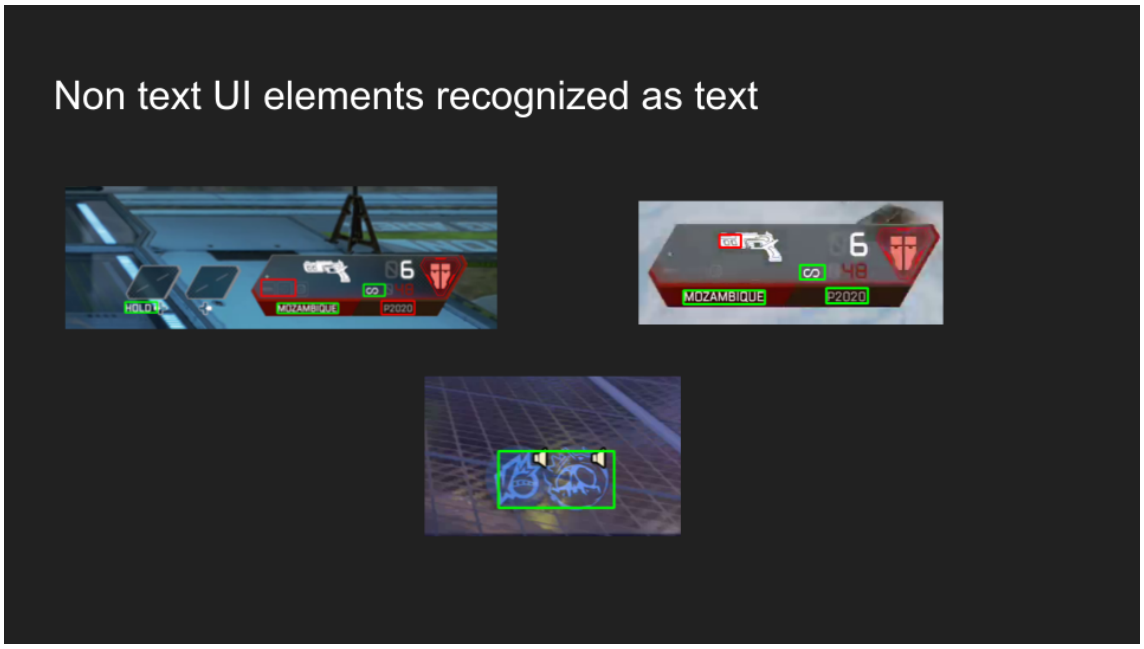


Figure 5.7: Non-text elements of the user interface detected as text.

As mentioned before, the tool has the possibility of being given a focus area to only run analysis in. This means that the environmental detection errors can be forgotten about, and with a bit more careful positioning of these regions of interests, the UI non-text elements detection can be skipped over as well.

All in all, these issues are not impeding the capacity of automation of the tool. Nonetheless, their workarounds do need a bit more attention to detail when testing massive amounts of data which means that, at the moment, the process cannot be completely automated.

5.3. Accuracy and reliability

In this section we will go over how each stage of the tool performs compared to a human being in terms of speed, exhaustiveness, correctness and dependability.

As mentioned in the previous evaluations, the tool is not perfect but it does present a big improvement in various areas of the accessibility validation process workflow that is currently in use.

5.3.1. Text detection reliability

The results show that the EAST algorithm is advanced enough to detect all the text on complex scenarios. Our solution presents a much more comprehensive analysis of the text in an image compared to what a human can do. How long would it take a human to extract every text box that is present in an image such as figure 5.3? For TinEye it does not matter if there are three or three thousand words in an image, it will analyze all of them with minimal time difference and without getting tired.

On the other hand, as mentioned in section 5.2.1, it is relatively common to get some text boxes that do not belong to the text we are interested in analyzing. This prevents full automation, since the evaluator cannot simply upload a series of images and just accept the result without checking for correctness. There are two approaches to solving this situation:

- Manually check that the detected text seems correct after processing. When getting the results from the analysis, a human would need to check the output images to see if the text boxes present in the image are correct. If the result was a failure due to a text box that did not belong to chat or subtitle text the tester would override the fail report and set it to correct if nothing else failed.

  Even though this process involves a manual check of every image, it greatly increases productivity as there is no need for a human to be measuring every word of the image. A person can determine at a glance whether failing text boxes pertain to the regions they are interested in.

- Determine regions of interest beforehand. If analyzing bulk media of the same game or with very similar positioning of the text the possibility of configuring focus regions comes in handy. The person who is setting up the test can simply indicate the regions where text will be and prevent environmental features to be detected. If running images in small batches this process could take some more time than checking for correctness at the end, but in the case of long videos or lots of images in different resolutions a tester could 'set it and forget it' and come back to get the final results.

5.3.2. Font size measurement accuracy

When following the manual procedure for measuring font size, a tester has to manually choose an appropriate word that they feel represents the font size based on *"hp"* distance (section 3.1.1), and measure the text.

This of course introduces human fallibility when no appropriate characters are present, also during the process of measuring the human can misjudge the edge of a character or simply misplace their cursor when measuring. Usually a two pixel error is admissible, but as the resolution increases, the relative margin of error this represents for a human decreases.

For example, when dealing with borderline compliant text, a 2 pixel error when using 720p resolution means a 11% error is acceptable since text height minimum is 18 pixels, but only a 4% error when dealing with the 52 height pixels required for 4k resolution text. As the videogame market moves towards higher resolutions, the margin of error that we can tolerate from human testers will become smaller and increase the need for a reliable text measurement solution.

TinEye ensures that its measurements always follow the same criteria: After applying the threshold mask to mark text it finds out the extreme positions of pixels pertaining to text and gets the dimensions from them. Where two different humans might judge differently where a character's edge starts and ends, the tool calculates the position based on edges that have enough contrast difference with the background.

5.3.3. Contrast check accuracy

The contrast calculations between text and background present the best improvement over humans from all stages described. The implemented solution improves the previous solution both in terms of accuracy and speed.

The manual procedure for measuring contrast required the tester to choose a singular relevant pixel from the background and another relevant pixel from the text to represent their colors. This introduces several possibilities for imprecision:

- Relevant pixel choice, both from text and background, can differ from tester to tester.
- This approach fails to take into account the whole background of the text. What happens if the background is a gradient or is striped with different colors?
- Detecting a difference in background significant enough to prompt another contrast check in a word near the one being tested can also vary from tester to tester.

TinEye's approach towards contrast calculation is more reliable and accurate. The comparison is no longer made on a per-pixel basis, the calculation is made between the average luminance of the whole text and the average luminance of its immediate surroundings. This calculation can be run on every piece of detected text, eliminating the need for choosing when it is relevant to be calculated.

5.4. Conclusion

TinEye represents an advancement over current manual techniques for accessibility guidelines validation in the video game industry. It is faster, more exhaustive and reliable with its measurements compared to its human counterparts. There are still some areas which can be improved upon, and it is not a solution for full automation yet. Some of the

issues can be worked around with the methods detailed above, but there is still room for improvement regarding speed and reducing the environmental detections in text detection.

In conclusion, even if not providing full automation, TinEye can be used to greatly speed up accessibility assurance for video games. This marks a step towards the possibility of every person being able to enjoy games regardless of their physical ability, making our industry a more welcoming environment for all the public.

Chapter 6

Conclusions and Future Work

And so, does the destination matter? Or is it the path we take? I declare that no accomplishment has substance nearly as great as the road used to achieve it. We are not creatures of destinations. It is the journey that shapes us.

The way of kings, Brandon Sanderson

There has been an increasing awareness in the video game industry around the world to make our games accessible to all kinds of players with different abilities so that everyone and anyone can enjoy the games we make. The current process used to check for these requirements is manual testing done by humans. A person cannot be expected to analyze image after image with text with dozens of words. They have to take a measurement they feel is representative to the rest of the text of the image and assume that if it passes, then the other instances of text around it do as well. Having a more exhaustive manual test is infeasible both in the volume of work it would represent and the time it would take to process it.

This project has described the design and development of TinEye, a tool that could automatically validate a set of accessibility guidelines for video games, specifically ones designed for color deficient vision and myopia. TinEye was created as a software that given a piece of media (such as an image or a video) would run a series of checks to ensure that it met certain accessibility criteria. The checks that TinEye runs are aimed mainly at chat and subtitle text in games but it can be used for other applications. It ensures that the text on screen complies with a minimum height required depending on the screen resolution and that the text color has sufficient contrast with its background so that it can be read clearly.

TinEye's development started with a series of prototypes to make sure that the idea and approach the team wanted to take towards this solution was achievable. During development agile development techniques such as Scrum and continuous integration testing were used to ensure that the tool was always working and that its design could adapt to changing needs from the team as well as techniques that could be found during research.

The most important part of the tool is finding out whether and where text is in an image. Various options were tested, ultimately arriving at EAST for text detection and OpenCV's solution for the recognition of said text. The tool was improved during many iterations, making it more reliable and accurate with its measurements than a human

would be.

This tool is a significant workflow improvement for the industry and will become even more relevant in the years to come. While full-automation couldn't be reached and manual revision of the reports is still needed, the throughput of a single person has been vastly improved. TinEye, as an assisted tool is capable of helping a tester much more productive than with a manual approach. Furthermore, even though it is mainly aimed at videogame accessibility guidelines validation, the flexible and generic nature of the tool makes TinEye able to work with any media, like GUI applications.

Future work will focus in improving the tools performance for deployment. It is expected that TinEye will be deployed as a back-end where EA testers and studios will upload the media to be analysed. Feedback from this deployment will be taken into account, focusing in providing the users all the information and features they need.

6.1. Future work

TinEye can be marked as a successful project for the objectives it set out to accomplish, but there are a number of improvements that could enhance and facilitate the accessibility validation process:

- Improve video processing speed by selecting specific frames. Execution speed could be increased by only selecting specific frames when analyzing a video. Video games, specially shooters or other fast-paced genres, have very frame rates, a high number of frames are being generated every second. This means that the content of sequential frames may be the same, moreso when talking about text which humans take time to read. This means that a lot of processing time is being spent on analyzing very similar frames.

  An algorithm can be developed that detects when a new frame to be processed is sufficiently different from the last one to be analyzed, if so it would run the accessibility checks on this frame and if not it would skip it.

- Implement a graphical interface for focus and ignore regions creation. The way that focus regions are currently defined in the configuration is by specifying the position and size of the rectangles relative to the screen size. For example, if the user wanted to define a focus region for the top left quarter of the screen they would have to state they wanted a region in the X dimension from 0 to 0.5 and likewise for the Y dimension.

  A graphical interface would make this process a lot more user friendly. After uploading the media, a sort of preview of the image could be shown and the user would draw the focus and ignore regions they wanted on top of this preview. This would later be translated to relative positions and input into the tool so it can function as normal. This way users and testers do not have to worry about how the underlying parsing of the rectangle definitions works.

- Train a specific neural network for text detection in a game scene. EAST was created to detect all kinds of text in real world scenarios, it expects to find instances of text in skewed and sometimes obscured situations. This is very practical for a system like a self driving car, but our tool's use case is different. Some of the 'accidental' environmental detections could be attributed to this fact.

Using the same algorithm but training a new neural network with a lot of training examples from different video game user interfaces can prove very beneficial in making TinEye more accurate and automating it.

- Create a real time overlay for testers. If sufficient performance is achieved, an overlay interface to inject into games could be created. This overlay would run the accessibility guidelines checks in the background and output in near-real-time if the text that is currently present on screen passes the guidelines' requirements.

  This approach could be useful while testing new interface configurations, since a keeping track of every text box result while playing organically could be difficult for a human and having to record the session and seeing the results after could be tiresome as well.

Bibliography

Greet every morning with a smile. That way it won't know what you're planning to do to it.

The Alloy of Law

Ashley Wood. The importance of colorblind accessibility in games. 2017. [Online; accessed 5-April-2022].

BenRG and cmglee. Comparison of some rgb and cmyk colour gamut on a cie 1931 xy chromaticity diagram. 2014. [Online; CC BY-SA 3.0].

Chernikov, Y. Basic instrumentation profiler. 2019. `https://gist.github.com/TheCherno/31f135eea6ee729ab5f26a6908eb3a5e#file-instrumentor-h`; Last accessed 28-03-2022.

Community, O. Opencv east algorithm tutorial. 2020. `https://github.com/opencv/opencv/blob/master/samples/dnn/text_detection.cpp` , last accessed 10-2-2022.

Community, T. Tesseract open source repository. 2022.

FCC. 21st century communications and video accessibility act (cvaa). 2010. [Online; accessed 6-April-2022].

Fredrick, D. R. Myopia. *BMJ*, Vol. 324(7347), 1195–1199, 2002. ISSN 0959-8138.

HDhes. Hdtv size and distance calculations. 2014. [Online; archived from original].

Joosten, E., Lankveld, G. v. and Spronck, P. Colors and emotions in video games. 61–65, 2010.

Martin Fowler. Continuous integration. 2006. [Online; accessed 27-April-2022].

Melillo, P., Riccio, D., Perna, L., Sanniti di Baja, G., Nino, M., Rossi, S., Testa, F., Simonelli, F. and Frucci, M. Wearable improved vision system for color vision deficiency correction. *IEEE Journal of Translational Engineering in Health and Medicine*, Vol. 5, 1–7, 2017.

Microsoft. Xbox accessibility guideline. 2021. [Online; accessed 5-April-2022].

Mithe, R., Indalkar, S. and Divekar, N. Optical character recognition. *International journal of recent technology and engineering (IJRTE)*, Vol. 2(1), 72–75, 2013.

MozillaCorporation. Web accessibility: Understanding colors and luminance. 2022. https://developer.mozilla.org/en-US/docs/Web/Accessibility/Understanding_Colors_and_Luminance", last accessed 15-03-2022.

Giuditta de prato, C. F. . J.-P. S. Innovations in the video game industry: Changing global markets. *Communications & Strategies*, 2014.

Sambasivarao, K. Non-maximum suppression (nms) a technique to filter the predictions of object detectors. 2019. https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c; Last accessed 4-4-2022.

Sharpe, K. R. G. . L. T. *Color Vision, From Genes to Perception*. 1999.

Sik-Lányi, C. 22 - choosing effective colours for websites. 600–621, 2012.

Smith, R. An overview of the tesseract ocr engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, Vol. 2, 629–633. IEEE, 2007.

Snook.ca. Snook.ca colour contrast check. 2015. Https://snook.ca/technical/colour$_contrast/colour.html fg = 33FF33, bg = 333333, lastaccessed$07/02/2022.

Tanaka, G., Suetake, N. and Uchino, E. Lightness modification of color image for protanopia and deuteranopia. *Optical review*, Vol. 17(1), 14–23, 2010.

Tom Wijman. The games market and beyond in 2021: The year in numbers. 2021. [Online; accessed 5-April-2022].

Tycho Henzen. What is the point of the colour blind filters in some games? 2021. [Online; accessed 5-April-2022].

W3C. How to meet wcag, guideline 1.4 – distinguishable. 2008a. [Online; accessed 6-April-2022].

W3C. Web content accessibility guidelines (wcag) 2.0. 2008b. [Online; accessed 6-April-2022].

W3C. W3c accessibility guidelines (wcag) 3.0. 2021. [Online; accessed 6-April-2022].

Wu, P.-C., Huang, H.-M., Yu, H.-J., Fang, P.-C. and Chen, C.-T. Epidemiology of myopia. *Asia-Pacific Journal of Ophthalmology*, Vol. 5(6), 386–393, 2016.

Yousefi, J. Image binarization using otsu thresholding algorithm. 2015.

Zhou, X., Yao, C., Wen, H., Wang, Y., Zhou, S., He, W. and Liang, J. East: An efficient and accurate scene text detector. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

Zhou, X., Yao, C., Wen, H., Wang, Y., Zhou, S., He, W. and Liang, J. East pretrained model. 2018. https://www.dropbox.com/s/r2ingd0l3zt8hxs/frozen_east_text_detection.tar.gz?dl=1 Last accessed 2022-02-20 .

Appendix A

Individual Contributions

A.1. Adrián Álvarez

During the beginning of the project outside EA, we had limited information about the current process. We knew that it would be required to emulate a humans capacity to find text, measure contrast and measuring text. My work focused, on researching the technologies and tools available regarding text recognition and contrast checking, as well as possible libraries for image and video manipulation like OpenCV. The results of this first contact meant having knowledge of the key technologies we were going to depend on during the development of the tool. During this prior research, I began to work on the state of the art chapter of this memory, taking note of why a particular technolgy is being considered and the possible limitations or pitfalls that could be found during the development.

Once the work inside Electronic Arts began, I spent some time learning the basics of CMake because one of the company's requirements for the tool was that it should be cross-platform. This process was slow in the start, the first prototypes were build as simple executables. Since the tool would need to be a library in order for it to be used as a backend for a future web application. Once development moved away from prototypes, the project was built as a static library and an executable that linked it. Lastly, regarding project configuration, I researched how to generate a test project with Google Test in order to develop and run tests locally. With some indications from company peers, I looked into vcpkg as a cpp dependency manager and used it to link the relevant dependencies such as a json parser for the configuration files, a logger, unit testing and OpenCV.

During the prototyping project my main focus, other that project building and configuration, was the EAST prototype, following the documentation and testing its capabilities with videogames. Regarding the contrast prototype I made sure that the formulas disclosed by Mozilla were the sames that were being used by the web currently being used in the manual workflow with Snook.ca (2015) and started familiarising with OpenCV for working with.

During the first iteration of the final tool, I started focusing on architecture, determining how classes would be exposed and what responsibilities would be left out to the final user or the library. For example, creating a Media factory that allows users of the library to load the media when they determine. I implemented a basic configuration class that

would be expanded as new features were developed and integrated the EAST prototype that would detect text for the following stages. To keep both of the team members in up to date, it was common practice to write tests for the other's implementations. I wrote some basic tests for size measurements and contrast calculations. The contrast calculation tests allowed us to find a precision error in the calculations even when using doubles. Esteban later fixed this issue by using look-up tables.

During the improvement iterations of the tool, I researched different profiling tools and instrumentalised the code to see how execution time was spent. This task prompted me to look into continous integration tasks. While the CI/CD pipeline had been provided by peers, I realised that it would be benefitial to run this profiling inside the pipeline. I looked into the already present implementation for automatically running tests and devised a similar job, where each time the source code from the library changed, TinEye would be run generating a downloadable artifact with the profiling data.

The profiling gave key information for the future of the tool development like the possibility of avoiding text recognition to increase speed. Another implemented optimisations were the filtering of tilted text boxes, which was proposed by Esteban, and the merging of aligned text boxes taking into account that all text would be aligned with the UI, decreasing the total number or text boxes.

Finally, in paralell to the optimizations, I worked with Esteban analysing the images that made up the final tests. I contributed to the deduction process of this tests, pointing out the main improvements that this tool is able to provide in comparison to the current workflow.

During the whole project I made sure to follow good agile practices: keeping an open communication with Esteban both to prioritise and estimate current work. I paid close attention in weekly meeting with the rest of the members of the team to find out what functionalies could make their work easier. This weekly meetings also helped the rest of the team to see the current state of TinEye, and I answered with all the details needed for them to plan around TinEye's capabilities. I reviewed each of Esteban's commits, to better comprehend the code that was going into the tool and that I would need to work with later.

A.2. Esteban Restrepo

In the investigation phase, prior to starting our work at Electronic Arts and the prototype development stage, I mainly focused on investigating different industry standards for contrast calculation and compliance requirements, and how different standards differed from each other. I also focused on finding out information on different OCR libraries and their possibilities, finally deciding on using Tesseract for our first character recognition tests.

During the propotype creation stage I created the basic contrast calculation prototype and the Tesseract OCR prototype. The contrast calculation prototype aimed to emulate the behavior of the website that EA's testers currently use to check contrast between two colors, but our tool received an image and two positions instead of two colors.

During this process I familiarized myself with the Tesseract API to make better use of it later down the line, although Tesseract would end up getting scrapped in our project. I also created various test cases for the contrast tool to make sure that we were getting the values we expected to get. These test cases actually gave use the first hints that Tesseract would not fit our needs, since gradient backgrounds with colored text would give lackluster results when ran through the OCR prototype.

Once development started at EA I focused on how to create the luminance map for the images input into the tool. I used OpenCV's matrix transformations to ensure that all processed images had three color channels, linearized all of them and multiplied each linearized channel using the luminance formula to obtain the final luminance value of a pixel. Finally, these values would be added up and encoded in a single channel matrix that represented the luminance map.

At this stage I also worked on build automation, utilizing CMake directives to recognize when new resource files had been added to the project and copy them automatically to the build folder when the project was compiled.

At first, all of the image information was inside the TinEye class alongside all of the checks and detection and recognition methods, making it very unwieldy to work on, so as the project grew I abstracted the image information and specific methods into the Image class (which would later turn be abstracted in the media class after video support was added) and documented our work throughout.

Afterwards, once the unit test framework was set up, I created various tests for size and contrast cases to make sure that the ongoing changes in our code would not break anything that was already working. The size tests consisted in passing and failing images each with text of one of five font types (serif, sans-serif, monospace, a bold font and an ultra-thin font) for each of the resolutions we developed the tool for (720p, 1080p and 4k). The contrast text cases consisted of a simple word on top of different backgrounds. The backgrounds could be either a solid color, a gradient, or stripes of different colors. two cases were made for each type of background, a passing one and a failing one, as well as high and low contrast test cases.

I created the Textbox class and used the now integrated EAST prototype to output a

list of regular rectangles that represent regions in an image where text is found. We could use these boxes for text measurement and later to highlight the output of passes and fails onto an image by coloring the boxes a certain color depending on the results.

After Tesseract proved to be ineffective for our use case, I researched and later implemented the alternative, OpenCV's text recognition solution. I made sure to integrate it well with our existing codebase and the better results from OpenCV were evident immediately.

I implemented the option to process video. Processing video proved to be tricky in a certain way: When processing images with OpenCV, depending on the filetype, the program can directly import them as matrices with specific properties, but when grabbing video frames you have to manually convert them to your desired matrix type. I had to implement certain checks during loading of a video frame to ensure correct conversion of the frame so the tool could run all of the subsequent tests and checks.

Later on I worked the possibility of calculating the luminance histogram of a region to improve the accuracy of our background contrast calculations, using them to find out the more significant values representing the colors of the text and its background. This option ended up not panning out since antialising and other noise in the image made histogram processing not really straightforward to work with.

To improve our contrast calculations I implemented the threshold and masking operations. I researched to optimal way to automatically calculate the threshold to divide an image into regions with text and regions with background, and how to use the resulting mask to create an outline of the text and get a more representative result of the text's background. This outline represents the background that just surrounds the text instead of the background of the whole text box as we had been doing up until that point. Finally, I created a battery of acceptance tests to ensure that our tool correctly passed or failed various font types in all of the supported resolutions and unit tests for our executive class TinEye to ensure correctness of the whole process.

Overall, during the whole process I worked closely with Adrian so we would always know the advances each one was making on the tool and we always had input on the design decisions of what was going on during development. I also attended various weekly meetings with different departments at EA to update them on the progress of the tool and get feedback on the features they would want TinEye to have.

The purpose of a storyteller is not to tell you how to think, but to give you questions to think upon

Wit
The way of kings
Brandon Sanderson