



S3 PROJECT
OPERATING FILE

OCR Word Search Solver

Group name:

DosRaieMythe

Albin Bocenno
Luc Desmottes
Ahmed Diallo
Maxime Robert

2024-2025

Contents

1	Introduction	2
1.1	About the project	2
1.2	Group Informations	2
1.3	Task distribution	2
2	Biography	3
2.1	Albin Bocenno	3
2.2	Luc Desmottes	3
2.3	Ahmed Diallo	4
2.4	Maxime Robert	5
3	Developpement	7
3.1	Albin Bocenno	7
3.2	Luc Desmottes	15
3.3	Ahmed Diallo	21
3.4	Maxime Robert	29
4	Additional tasks	36
4.1	Character Generator	36
4.2	Website	37
4.3	Word Search Generator	38
5	Conclusion	40

1 Introduction

Welcome to this project report! In this document, we will discuss the development of the OCR Word Search Solver project. While we won't delve too deeply into technical details, we will provide a brief overview of the project's development. We'll cover the group members, task distribution, challenges encountered, and more.

1.1 About the project

The objective of this project is to primarily familiarize ourselves with image processing, character recognition (OCR), and the functioning of neural networks. The project focused on solving Word Search puzzles by processing an image, utilizing a neural network, and implementing a solver algorithm to return the identified words.

1.2 Group Informations

The group is made up of four people: the group leader Albin Bocenno, along with Luc Desmottes, Ahmed Diallo, and Maxime Robert. We are all in the same A1 class in the English-speaking program.

1.3 Task distribution

After the formation of the group, we divided the tasks as follows: Luc was responsible for coding the solver algorithm, Ahmed was in charge of the image rotation, Maxime handled the image binarization, and Albin focused on the word grid detection and the neural network.

Later in the report, each member will explain the progress of their development work.

2 Biography

2.1 Albin Bocenno

I discovered computer science through cybersecurity. After a high school observation internship with the COSMIQ team, a group of researchers specializing in post-quantum cryptography, I immediately wanted to learn more about cybersecurity. This curiosity led me to programming, and over time, I began developing small projects in Python. Each project allowed me to deepen my understanding and gain practical skills, sparking a growing interest in exploring more complex aspects of programming and digital security.

When I arrived at EPITA, I was able to continue coding and learn more about different languages like C#, OCaml, and finally C. Before EPITA, I knew nothing about C or dynamic memory management, and I really enjoyed the fact that it is a low-level language.

I find that this project is an excellent way to learn new things, such as neural networks, image processing, and more.

2.2 Luc Desmottes

My journey into computer science began early, when I was bored and decided to explore a small program called Scratch 1. As I delved deeper into it, I became increasingly curious about the possibilities it held. Over time, I outgrew Scratch and sought to challenge myself further, progressively moving to languages like Python, C#, and now, C. This natural inclination to push myself technologically led me to pursue a degree in computer science.

Upon arriving at EPITA, I was certain that I would specialize in either cyberse-

curity or artificial intelligence. However, my interests have expanded to encompass various fields within computer science. For instance, I have discovered a strong passion for software engineering and front-end development. Additionally, I have been engrossed in a personal video game project that aligns with my desire to continuously challenge myself.

The OCR project and the S2 project from last year have further fueled my exploration of potential career paths. In addition, the OCR project allows me to explore user facing interfaces, be it by rendering a clean image at the end of the project, or by making a user friendly application. During these projects, I believe that collaborating with a team is crucial for delivering a functional and high-quality piece of software within the limited timeframe.

2.3 Ahmed Diallo

My interest for computer science comes from my passion for video games and my mother. Indeed, I have been playing video games ever since I was a child so I really wanted to know how it can work. My mom graduated from EPITA (in Telecom), hence I followed her steps and went to EPITA to pursue my education in computer science.

Artificial intelligence is a new emerging technology that is taking more and more space in our lives so I am interested in knowing how it work. Hence the OCR project is interesting and can help me understand a little bit how it works. These new acquired skills will help me learn more about Artificial Intelligence and will be helpful in the future as I will pursue my studies in this sector. The group aspect of the project will also be useful later since in computer science it is impossible to do everything by ourselves hence each person will do a part and work together in order to be as efficient as possible.

The deadlines being short and closing in rapidly, we have to be efficient and work together with a good organization or the future of the project will be uncertain.

2.4 Maxime Robert

I became interested in computer sciences thanks to video games. Being a passionate fan of video games, I spend a significant amount of time on my computer. By using my computer so frequently, I eventually began to wonder how it functioned and how I could resolve the various technical issues I encountered.

Over time, I developed an interest in programming. My passion for programming and computers in general motivated me to pursue studies in computer science. I am primarily interested in two areas within computer science: artificial intelligence and video game design. Artificial intelligence fascinates me because it is still a relatively new field and is constantly evolving. Video games, on the other hand, have captivated me since I was young, and I would love to be part of this industry and contribute to what I consider the pinnacle of human artistry.

The OCR project aligns perfectly with my interest in artificial intelligence, and the image preprocessing aspect can be connected to graphic programming, which is used in video game creation. This project allows me to explore some aspects of computing that I will study in greater depth later in my academic career.

The teamwork aspect is also crucial, as in computing, it is nearly impossible to accomplish everything independently due to the complexity of the field. The very short deadlines for the project compel us to organize ourselves effec-

tively and efficiently in order to minimize time loss and to begin the project as soon as possible, with the goal of delivering a product of the highest possible quality.

3 Developpement

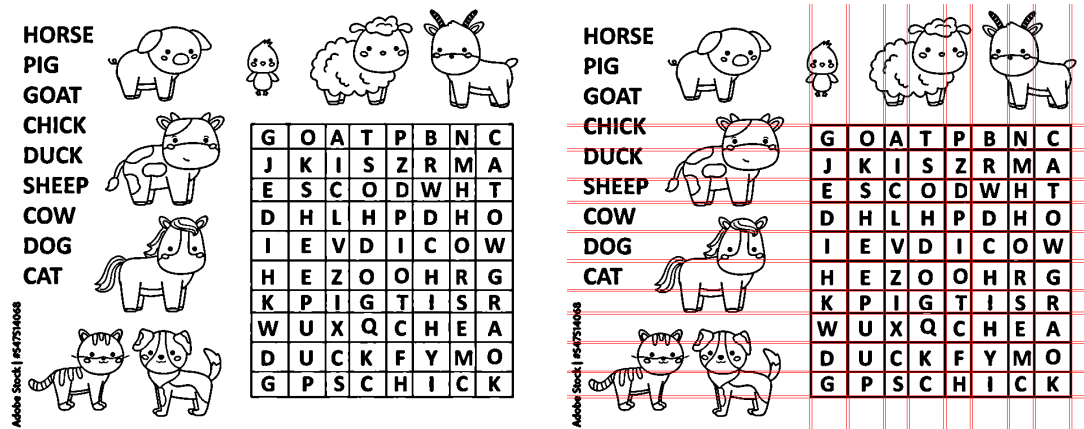
3.1 Albin Bocenno

I first worked on the detection of the grid and the list of words. For the detection of the grid, I searched online for ways to detect lines in an image. A simple Google search for "How to detect lines in an image?" shows that the "Hough Transform" is the most well-known method for detecting lines.

The process of the algorithm is quite simple to understand and code. We assume that the image is in black and white, meaning that a pixel is either black or white.

The first step is to find the edges. An edge is a pixel where there is a sudden change in color. We can use several algorithms to detect edges, such as the Laplace filter, the Sobel operator, or the Canny algorithm. I opted for the Canny algorithm because I found it simpler and more effective. Without going into details, for each pixel the Canny algorithm looks at the pixels on the left and right, and then above and below, and sums them up according to a specific matrix. With this, I can detect the edges.

Next, I can proceed to the second step; by traversing the edges, I store all the possible lines crossing the edge in a list in polar coordinates. Thus, at the end of my traversal, the peaks in my list represent the lines with the most edges placed on. An example of hough transformation:



The problem is that not all images necessarily have a grid with well-defined lines. In some images, there is a grid of letters without the lines that delimit these letters. For a while, I thought it was always possible to find these invisible lines by lowering the threshold and refining my algorithm that searches for peaks in my accumulation list. However, the problem became too complex, so I abandoned that approach and only kept Hough for the few images where it worked.

I then thought of another algorithm. Instead of searching for lines, I could look for groups of pixels. The process is relatively simple: for each black pixel encountered in the image, I search for all the surrounding black pixels and add them to a structure called Letter. Having recorded the position of each pixel, it becomes relatively easy to eliminate the grid (the height of the Letter being too great or too wide), as well as some noise since the letter has too few pixels. By increasing the tests, I finally managed to find all the letters in the image, except for two images.

The second problem is now how to differentiate the letters in the grid from the letters in the word list. If my Hough algorithm worked, then it's easy to detect this. All letters inside the grid are letters from the grid. But when Hough didn't work, I had to use another approach.

I employed two tests that work in most cases: the first being that the letters in the grid are more spaced out than the letters in the word list. A simple average of the shortest distance from one letter to another allows us to separate those in the list from those in the grid. For the few small letters detected as being in

the grid but outside of it, I perform another test on the list of letters in the grid. I accumulate the x and y axes for each letters in the grid. Since the letters in the grid are aligned along the x and y axes by more than 8 times, those with an accumulation axis in x or y less than 8 are isolated from the others, thus indicating that this letter belongs to the word list.

By going through the numerous tests I had to add to my Hough algorithm or the letter detection algorithm, here are the few results I obtained (one failed !).

MINDFULLNESS	P X U T S I N I U P R V G B M D D
IMAGINE	E H A A S P O J P E T B E Q Z L C
RELAX	A U N T E G Q T L H R Z F A T O P
COOL	S H X F N G U A X E A A Y P O M H
RESTING	Y O Y Y L D X L A K Y U Z L B S K
BREATHE	J X M U U G Q T R I M A G I N E B
EASY	H F N W F X H D P B B B T N V S K
TENSION	H I I H D E S Q F U M Y E R N S X
STRESS	R P B Z N H S D S L H O N B S S S
CALM	E H X A I Z I H A H O E S Q F E F
	C W Z I M V D C J V S S I M G R W
	L A I I R Z Q Q H X D Z O Z Q T R
	W C A X E Z R G H A I Z N E C S E
	B R H F O T G N I T S E R E O V Z
	M W V W Q D U I H W Q T S B I M L
	T D T O N Z C X X R G E L K H F Q
	Q N E K S V M O T F A L A A E W B

M	S	W	A	T	E	R	M	E	L	O	N
Y	T	B	N	E	P	E	W	R	M	A	E
R	R	L	W	P	A	P	A	Y	A	N	A
R	A	N	L	E	M	O	N	A	N	E	P
E	W	L	E	A	P	R	I	A	B	P	R
B	B	I	L	B	B	W	B	R	L	A	Y
K	E	M	P	M	A	W	L	R	A	R	B
C	R	E	P	R	N	R	E	R	R	G	R
A	R	Y	A	Y	A	O	A	N	L	A	M
L	Y	Y	A	R	N	E	R	K	I	W	I
B	E	B	A	A	A	N	A	A	P	R	T
Y	R	R	E	B	P	S	A	R	N	N	W
Y	R	R	E	B	E	U	L	B	L	G	I
T	Y	P	A	T	E	A	E	P	A	C	E

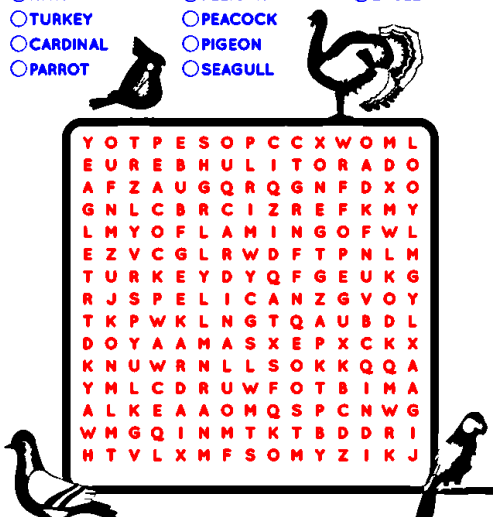
APPLE
LEMON
BANANA
LIME
ORANGE
WATERMELON
GRAPE
KIWI
STRAWBERRY
PAPAYA
BLUEBERRY
BLACKBERRY
RASPBERRY

Find the words below and circle them (across, down or diagonally)

☐ FLAMINGO
☐ KIWI
☐ TURKEY
☐ CARDINAL
☐ PARROT

☐ TOUCAN
☐ PELICAN
☐ PEACOCK
☐ PIGEON
☐ SEAGULL

☐ SWAN
☐ EAGLE



Y	O	T	P	E	S	O	P	C	C	X	W	O	M	L
E	U	R	E	B	H	U	L	I	T	O	R	A	D	O
A	F	Z	A	U	G	Q	R	Q	G	N	F	D	X	O
G	N	L	C	B	R	C	I	Z	R	E	F	K	M	Y
L	M	Y	O	F	L	A	M	I	N	G	O	F	W	L
E	Z	V	C	G	L	R	W	D	F	T	P	N	L	M
T	U	R	K	E	Y	D	Y	Q	F	G	E	U	K	G
R	J	S	P	E	L	I	C	A	N	Z	G	V	O	Y
T	K	P	W	K	L	N	G	T	Q	A	U	B	D	L
D	O	Y	A	A	M	A	S	X	E	P	X	C	K	X
K	N	U	W	R	N	L	L	S	O	K	K	Q	Q	A
Y	M	L	C	D	R	U	W	F	O	T	B	I	M	A
A	L	K	E	A	A	O	M	Q	S	P	C	N	W	G
W	H	G	Q	I	N	M	T	K	T	B	D	D	R	I
H	T	V	L	X	M	F	S	O	M	Y	Z	I	K	J

For the proof of concept of the neural network, I looked at various webpages to understand the steps involved in creating a neural network. At first, I found it a bit confusing and very complex. However, for a simpler network like the one required for the first presentation, the operation is quite straightforward. It requires two inputs and one output to represent the following logical function $\overline{A} \cdot \overline{B} + A \cdot B$.

First, I defined what a neuron is in my program. Each neuron has two important parts: weights and a bias, which help it make calculations.

```
typedef struct {  
    double weights[2];  
    double bias;  
} Neurone;
```

Next, I set up the neurons by giving them random starting values for their weights and bias. Then, I wrote a function that calculates what the neuron outputs when it receives input data. During training, I fed the neurons pairs of numbers as input, checked what they produced, and compared that to the expected answer. If the output was incorrect, I adjusted the weights and bias to improve the result.

Console output from the neural network:

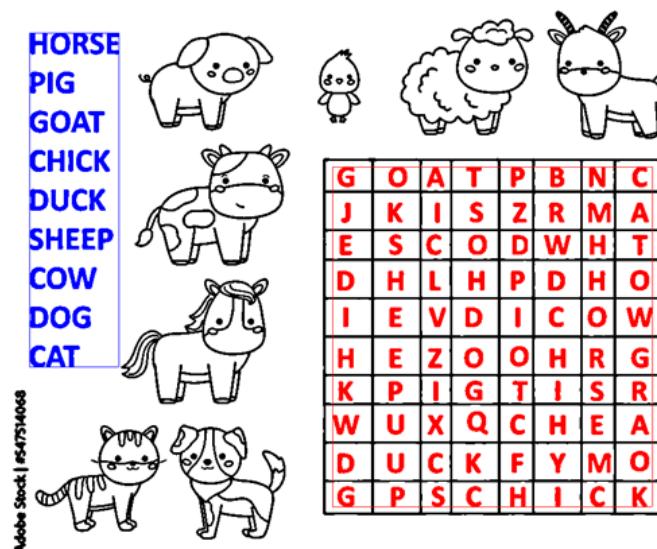
```
Results after training :  
Input: [0.0, 0.0] Expected output: 1.0 Output: 0.9960692887  
Input: [0.0, 1.0] Expected output: 0.0 Output: 0.0033028281  
Input: [1.0, 0.0] Expected output: 0.0 Output: 0.0033028296  
Input: [1.0, 1.0] Expected output: 1.0 Output: 0.9966419366
```

I have an activation function that helps determine, by calculating the derivative, whether the error is significantly far away and requires more adjustment, or if it is very close and therefore needs only a small change. I came across several

activation functions, such as the tangent or ReLU functions, but I found that the sigmoid function was the most commonly used, so I decided to implement it.

I worked on the automatic rotation of the image. Since I was using the Hough transform for character detection, my program simply observed how many degrees the lines deviated from the horizontal and vertical axes to rotate the image accordingly. I then improved my algorithm for letter detection. Among all the tests I added, I added the variance in letter sizes to eliminate noise among the letters in the word list.

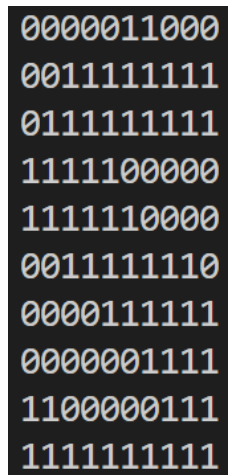
By calculating the height of all the letters in the word list, we notice that the height differs by at most a few pixels. We can easily detect noise if its size differs significantly. In the end, for all the images, my program detects the letters in the grid and for most of the images, the letters from the word list without the noise.



After that, I completely built the neural network. I started by working on the preprocessing of the letters. I arranged them in the order they appeared in the

image, easily detected the grid dimensions and the number of letters per word in the list, and then scaled them into matrices of 0s and 1s.

I hesitated for a long time when deciding on the size of the matrix, but I eventually opted for a reasonable size of 10 by 10.



```
0000011000
0011111111
0111111111
1111100000
1111100000
0011111110
0000111111
0000001111
1100000111
1111111111
```

By storing these matrices in a text file and manually writing the corresponding letters, I was able to load the data into my neural network and train it.

As for the network, I decided to reuse the one developed for the non-XOR function. In fact, I only changed the input size (a list of 100) and the output size (a list of 26 representing the index of the letter in the alphabet). I kept the sigmoid activation function and my training function. I conducted numerous experiments to determine the optimal number of neurons, layers, and training iterations for my code. In the end, I found that 1,000 training iterations with two layers of 1,000 and 500 neurons, respectively, were more than sufficient.

The final challenge was the training time—it took several dozen minutes to complete. To address this, I divided the network into two distinct parts. The first part was the training phase, where the network processed an input file

containing the matrices and their expected outputs to train itself. At the end of the training, it saved the neuron weights and biases into a file named neurons.txt.

The second part (used after the network was trained) loaded the neurons.txt file (which only took a few seconds) and used the stored values to process input matrices and return the results. This separation made the process much more efficient for practical use. Indeed, in advance, I train the network (which can take an hour or more) and save the results in my file for practical use.

With the help of the solver, we were able to detect multiple words from several images. By retrieving the position of the letters, I managed to draw lines to display the result on the screen.

However, one issue persisted: the binarization process often caused several letters to merge together, making it impossible for the network to correctly identify the overlapping letters. This limitation significantly impacted the accuracy of the system, particularly in cases where the letters were tightly spaced or poorly segmented during preprocessing.

**APPLE
LEMON
BANANA
LIME
ORANGE
WATERMELON
GRAPE
KIWI
STRAWBERRY
PAPAYA
BLUEBERRY
BLACKBERRY
RASPBERRY**

In theory, I could have increased the size of the matrix to store the data of the merged letters, but this would have been impossible in terms of training time. I think it would have required several days of training to achieve satisfactory results.

3.2 Luc Desmottes

The algorithm responsible for solving the problem is methodically executed in multiple stages, each serving a distinct purpose in identifying the target words within a grid. The first critical step involves locating the initial letter of the word in question. This process requires an exhaustive sweep of the grid to pinpoint all potential starting positions that match the first character of the word. This is a necessary step because, without identifying the starting letter, it is impossible to initiate the search for the full word.

Once potential starting points have been identified, the algorithm proceeds to the second stage: exploring all possible directions from each starting point. In a two-dimensional grid, this involves checking eight potential directions—up, down, left, right, and the four diagonals. The algorithm evaluates whether the sequence of letters in any of these directions matches the word we are searching for. This step is crucial for ensuring that the word is correctly found and aligned, whether it is positioned horizontally, vertically, or diagonally within the grid.

An essential aspect to keep in mind is the case sensitivity of the search. The target words may not be presented in uppercase letters, even though the grid itself is entirely in uppercase. This discrepancy demands a careful comparison between the grid and the words, ensuring that the algorithm correctly identifies matches without being misled by differences in letter case.

```
char solve(char**grid, size_t width, size_t height, char* word,
           size_t* o_startCol, size_t* o_startRow, size_t* o_endCol, size_t* o_endRow);
```

However, the current implementation of the system introduces additional challenges. One of the main issues is the complexity arising from the presence

of numerous parameters that need to be managed during the search process. This includes an entire array of output values whose pointers must be fed to the algorithm, which adds layers of intricacy to the use of the algorithm. Such complexity can make the methods less user-friendly and harder to maintain or modify. The management of these parameters may require developers to navigate through extensive and

```
char compareChars(char a, char b)
{
    if(a >= 'a' && a <= 'z')
    {
        a = a - 'a' + 'A';
    }

    if(b >= 'a' && b <= 'z')
    {
        b = b - 'a' + 'A';
    }

    return a == b;
}
```

potentially confusing code structures, thus impeding the ease of use and clarity of the solution. Furthermore, the standalone ‘solver’ program necessitates additional steps, primarily involving the loading of the grid from a file. Given that the grid’s size is unknown to the program, I commence by reading the grid’s size. Subsequently, I restart the file read operation and populate the allocated grid with characters from the file. This approach is employed to avoid repetitive memory reallocations on the heap, as the latter operation is not the fastest.

For the User Interface (UI), we initially considered various frameworks and tools before deciding not to reinvent the wheel from scratch. We opted to use Gtk, a well-established and widely used toolkit, to handle the interface el-

ements. Gtk provides robust features for creating graphical interfaces, which would save us time and effort compared to building everything from the ground up.

Our process began with extensive pen-and-paper sketches, where we brainstormed the layout, interactions, and overall design of the User Interface (UI). This initial phase was crucial in shaping the direction of the project, as it allowed us to visually explore different ideas and concepts before committing to any specific design. The sketches were not just rough drafts, but detailed representations that included key elements such as buttons, navigation flow, and the arrangement of content.

By sketching out various ideas, we were able to quickly test different approaches to the UI, assessing which design would best suit the user's needs and the goals of the application. It also helped us identify potential usability issues early on, such as confusing navigation or unclear visual hierarchies. This way, we could refine and iterate on the design before starting any development.

The benefit of working on paper first was that it gave us the freedom to experiment and iterate without the constraints of software or code. We could easily modify the design, shift elements around, and experiment with new features. This process of sketching allowed for rapid prototyping, which ultimately led to a more polished and well-thought-out final design.

Moreover, this step ensured that we had a clear visual roadmap for the app before we moved into the digital realm. Once we had a solid design foundation on paper, we were able to translate it into code with confidence, knowing that we had already refined the structure and flow. This approach helped us stay focused on the core principles of the UI, such as usability and clarity, and made it easier to adapt and improve the design during the actual development

process.

Once we had a clear vision of the UI, we transitioned into the development phase. We began by writing fundamental functions that created simple interface components such as buttons, labels, and images. As we progressed, we began developing more complex elements like menus, dialogs, and dynamic content that responded to user actions.

Throughout the development process, we focused on incremental progress, continually refining and adding new elements to the UI until we had the final layout. By building functions in a modular way, we ensured that each component was reusable and easy to update in the future. This approach allowed us to stay organized and maintain control over the design and functionality, making it easier to implement changes and fine-tune the user experience as we moved forward.

The User Interface needs to integrate seamlessly with all the systems that have been developed so far. Since we've already built various components and functionality using SDL, it's crucial that the UI can interact with these existing systems. One challenge that arose during this process is that the image handling mechanisms used in the SDL-based systems rely heavily on `SDL_Surface`, which is the standard structure for handling pixel-based images in SDL.

However, Gtk uses its own image handling system, which is different from SDL. Specifically, Gtk works with its own image format and structures, such as `GtkImage`, which is not directly compatible with `SDL_Surface`. This discrepancy presented a potential roadblock, as the existing systems we had developed so far expected images in SDL format, but Gtk expects them in a different format.

To address this issue, we need to bridge the gap between these two image handling systems. This could involve creating a conversion mechanism that allows SDL images (SDL_Surfaces) to be transformed into a format that Gtk can work with, or vice versa. This conversion process would ensure that all the graphical elements, such as images, textures, or even more complex visual data, can be displayed properly within the Gtk interface without breaking the connection to the already-developed SDL-based systems.

To address the challenge of integrating SDL-based systems with Gtk, we implemented a solution that allows us to convert images from SDL_Surface (used in our existing systems) to Gtk images (which are required for our User Interface). This conversion was essential because SDL and Gtk have different image formats and structures, and without this bridge, it would have been difficult to display our existing content in the new UI.

The implementation involved creating custom functions that can take an SDL_Surface, which stores pixel data in a specific format, and convert it into a Gtk-compatible image format, such as GtkImage. These functions handle the conversion of pixel data, color formats, and any necessary adjustments to ensure that the image displays correctly in the Gtk environment. Our approach focused on efficiency to ensure that the conversion process does not slow down the app or compromise performance. We optimized the conversion process by carefully managing memory and ensuring that image data is copied in a way that minimizes overhead.

By implementing this conversion functionality, we were able to ensure seamless integration between the existing SDL-based systems and the new Gtk-based UI. This allowed us to leverage the image handling capabilities of both systems, without needing to overhaul the entire image infrastructure or rewrite the existing systems. Ultimately, this solution made it possible to maintain the functionality of the SDL components while meeting the requirements of Gtk,

ensuring that the user interface could display all the necessary images without issue.

3.3 Ahmed Diallo

At first, we tried to use integrated function of the SDL Library. But the main issue was that the renderer was not showing the whole image because the dimension changes and also the SDL surface of the new image was not change hence it couldn't be used so we had to create a new algorithm that take care of the rotation pixel by pixel.

The image rotation feature we developed tackles the challenge of transforming an image by a specified angle while preserving the visual quality of the image and minimizing distortions or loss of data. Our approach involves several stages, each crafted to address specific technical aspects of image rotation, from boundary calculation to pixel interpolation. The primary goals guiding this process were ensuring clarity, maintaining visual fidelity through precise interpolation, and optimizing for efficiency.

One of the core challenges in image rotation is accurately determining pixel values for points that do not align directly with integer pixel coordinates in the rotated image. The rotation transformation often results in fractional coordinates that fall between pixels, making it essential to approximate these values smoothly to avoid jagged or “stepped” edges.

To achieve this, we implemented a bilinear interpolation function, bilinear interpolation. This function takes advantage of the four nearest neighboring pixels to estimate the value at any non-integer coordinate. The process begins by identifying the integer coordinates immediately surrounding the target point. Then, fractional distances dx and dy —representing how far along the x and y axes the target point lies between these neighboring pixels—are calculated. These fractions serve as weights in a linear blend, or interpolation, for each color channel (red, green, blue, and alpha).

The bilinear interpolation computes the color by performing two interpolations along the x -axis, blending values horizontally between pairs of neighboring

pixels, and then a final interpolation along the y-axis. This two-step process yields a smooth, continuous color transition, reducing the "staircase" artifacts typically associated with rotated images. The advantage of bilinear interpolation lies in its simplicity and effectiveness for achieving a visually coherent output without compromising efficiency. Each pixel color is a weighted blend, contributing to the overall smooth appearance of the rotated image.

Rotating an image does not simply involve rotating the pixels; it also changes the rectangular boundary of the image. As the image is rotated, parts of it may extend outside the original boundaries, requiring additional space to avoid clipping. The `calculate_new_dimensions` function calculates the appropriate new width and height to contain the entire rotated image. This function uses trigonometric principles to ensure that no part of the rotated image falls outside the new boundaries, regardless of the angle.

The calculation involves converting the input angle from degrees to radians, which is necessary since the trigonometric functions `sin` and `cos` expect angles in radians. The sine and cosine values of the angle are used to compute how much the width and height of the original image will expand in both x and y directions. Specifically, it computes the absolute values of the projections of width and height onto the rotated axes. This results in the minimum width and height needed to fit the rotated image within a new rectangular bounding box.

This boundary calculation step is crucial in ensuring a complete, undistorted image rotation. By recalculating the dimensions before performing the rotation, we avoid scenarios where parts of the image might be cut off or distorted due to insufficient space, thereby maintaining a high-quality result.

The `rotateman` function serves as the primary rotation function, orchestrating the steps to create a rotated version of the image. This function takes the source image and the specified rotation angle as inputs, calculating the neces-

sary adjustments and creating a new surface with the dimensions calculated by `calculate_new_dimensions`.

The function then establishes a central reference point, calculating the image's center coordinates in both the original and the newly created surface. This approach ensures that the rotation is centered around the midpoint of the image, creating a visually balanced output. The function also precomputes the trigonometric values (sine and cosine) of the rotation angle to optimize performance, reducing the need for repeated calculations within the nested loops that process each pixel.

For each pixel in the target surface, the function maps its coordinates back to the source surface using the inverse rotation transformation. This transformation formula converts destination coordinates (x, y) in the rotated image back to the corresponding source coordinates $(srcX, srcY)$ in the original image. This mapping is essential because it allows us to draw each pixel in the rotated image by looking up the correct location in the original image, ensuring an accurate rotation.

The mapped source coordinates are typically non-integer values, especially in diagonal rotations, requiring the bilinear interpolation function discussed earlier. If the calculated coordinates fall outside the bounds of the original image, the function defaults to assigning an out-of-bound color, in this case, white (or transparent if desired). This boundary handling ensures that areas outside the rotated image's bounds do not produce artifacts or random colors but instead have a uniform appearance.

The main program, found in `main.c`, is structured to handle image loading, processing, and rendering through SDL (Simple DirectMedia Layer), a multimedia library well-suited for graphics manipulations. At the start, it initializes SDL and `SDL_image` libraries, which allow for handling various image formats

and graphical operations. The main program then accepts command-line arguments for specifying the image file and rotation angle, providing a flexible interface for testing different rotation angles on various images.

Once an image is loaded, the program invokes `rotateman` to perform the rotation with the desired angle. It then adjusts the window size to match the dimensions of the rotated image and displays the result. Additionally, the program saves the rotated output as `rotated.png`, allowing for further inspection and validation of the rotation results.

To visualize the image, a new SDL texture is created from the rotated surface, which is then displayed in the window created by the SDL renderer. This workflow ensures that users can see the result of the rotation and provides a simple interface for further image processing or analysis. The program also includes an event loop that listens for quit events, enabling users to keep the window open for as long as necessary.

In summary, our implementation provides a robust and efficient solution for rotating images by arbitrary angles, preserving visual fidelity through the use of bilinear interpolation and maintaining structural integrity through accurate boundary calculations. This approach successfully addresses common challenges in image rotation, such as pixelation, clipping, and computational inefficiency. The result is a solution that produces high-quality rotated images, adaptable for further processing or analysis, and suitable for applications where accuracy and clarity are paramount. This technique not only demonstrates an understanding of image processing principles but also optimizes performance to provide real-time, visually coherent results.

Optimization was a cornerstone of our approach to implementing the image rotation functionality, as we aimed to balance high-quality output with efficiency. Throughout the development process, we identified several bottlenecks and

implemented strategies to overcome them, ultimately achieving a solution that is both robust and performant. Below, we detail the optimization techniques applied at various stages of the algorithm and their impact.

One of the most computationally intensive operations in image rotation is the calculation of trigonometric functions, specifically sine and cosine. These functions are used repeatedly in the rotation transformation to map destination pixels back to their corresponding source coordinates. Recomputing these values for every pixel in a loop would have introduced significant overhead, especially for larger images. To address this, we precomputed the sine and cosine of the rotation angle once at the start of the process. These values were then reused throughout the algorithm. This seemingly simple optimization had a profound impact on performance, as it drastically reduced the number of function calls and associated computations during pixel mapping.

Memory management is another critical aspect of optimization, particularly when dealing with images that can occupy large amounts of space. Our algorithm dynamically allocated memory for the rotated image based on the dimensions calculated by the `calculate_new_dimensions` function. By determining the exact size of the bounding box required to fit the rotated image, we ensured that no unnecessary memory was allocated. Additionally, we avoided frequent memory reallocation, which can lead to fragmentation and increased processing time. This approach not only conserved resources but also sped up the overall execution by ensuring efficient use of system memory.

The process of mapping each pixel in the destination image back to the source image is inherently iterative and can become a bottleneck if not handled carefully. In our implementation, we optimized this step by ensuring that each target pixel was processed in a single pass. The algorithm calculates the source coordinates for each target pixel, applies bilinear interpolation (if within bounds), and assigns the corresponding color value in one operation. By

avoiding redundant calculations and unnecessary iterations, we reduced the complexity of the loop, leading to a noticeable improvement in processing speed.

SDL (Simple DirectMedia Layer) provides a suite of highly optimized functions for handling surfaces and pixels. Instead of reinventing the wheel, we integrated these functions into our workflow to capitalize on their efficiency. For example, SDL's built-in functions for accessing and modifying pixel data enabled us to perform low-level operations quickly and reliably. This allowed us to focus on higher-level optimizations, confident that the underlying operations were being handled as efficiently as possible.

Bilinear interpolation was chosen as the method for calculating pixel values at fractional coordinates. This decision was based on its ability to deliver smooth, visually pleasing results without the computational intensity of higher-order interpolation methods. By performing two linear interpolations along the x-axis, followed by one along the y-axis, we achieved a balance between computational efficiency and output quality. This method is particularly well-suited for real-time applications, as it avoids the jagged edges that can result from simpler nearest-neighbor interpolation while remaining faster than cubic or spline-based methods.

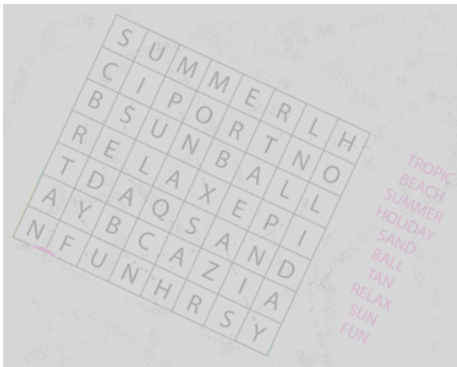
An often-overlooked aspect of optimization is the handling of edge cases, such as pixels that map to coordinates outside the bounds of the source image. In our implementation, such pixels were assigned a default color (white or transparent). This straightforward approach not only ensured visual consistency but also avoided the need for additional checks or complex fallback logic, simplifying the algorithm and improving its performance.

The `calculate_new_dimensions` function is an integral part of our optimization strategy. By pre-computing the dimensions of the bounding box required

to fit the rotated image, we avoided the need for costly trial-and-error or iterative resizing. This calculation used trigonometric projections to determine the maximum width and height of the rotated image, ensuring that all parts of the image would fit within the new dimensions. This step was particularly important for reducing the computational load during the actual rotation process, as it allowed us to allocate memory and process pixels with precise bounds in mind.

Optimization is an iterative process, and profiling played a key role in identifying areas for improvement. By using profiling tools to measure the execution time of different components of the algorithm, we were able to pinpoint bottlenecks and prioritize optimizations. For example, early profiling revealed that trigonometric calculations and memory allocations were consuming a disproportionate amount of time. Addressing these issues through precomputation and efficient allocation led to significant performance gains. Similarly, profiling helped us fine-tune the parameters of the bilinear interpolation function, striking the right balance between speed and visual quality.

Finally, our optimizations were guided by the principle of scalability. Although the algorithm was designed for standalone rotation tasks, it is also capable of handling real-time processing requirements, such as those found in live image streams or interactive applications. By focusing on reducing the computation time per pixel and optimizing memory usage, we ensured that the solution could scale gracefully to larger images or more demanding use cases.



3.4 Maxime Robert

The binarization of the image is complete. Although our initial goal was to achieve a perfect binarization, we found that identifying an algorithm that was both efficient and user-friendly proved more complex than anticipated. Despite this, the results from our current algorithm remain satisfactory and provide a strong foundation for moving forward. We would have preferred to achieve a near perfect binarization where no information would be lost, but we could not find a way to get an output of such quality. This refined output would have been crucial for delivering high-quality data to the neural network, supporting accurate and reliable results, even when processing images with significant noise.

For the image preprocessing, we divided our tasks into several stages: grayscale conversion, noise reduction, and finally, black-and-white transformation. Before implementing an algorithm, we needed to convert the image to grayscale to make it easier to process; in grayscale, each pixel is represented by a single value between 0 and 255, unlike RGB which requires three values.

To convert our image to grayscale, we use a grayscale formula, which takes as input the red, green, and blue values of a pixel and outputs a gray value between 0 and 255. This formula performs a weighted sum of these three values, with the weight assigned to each color based on how the human eye perceives brightness.

For example, the human eye perceives green more strongly than blue or red, so the green value is given a higher weight (0.7152).

A considerable amount of time in the image preprocessing phase was dedicated to researching and evaluating various binarization algorithms to convert the image to black and white effectively. We started by implementing a basic, self-designed algorithm to gain a more hands-on understanding of the challenges involved. This algorithm divides the image into small segments and

calculates the average brightness within each segment. Pixels with brightness below this average are turned black, while pixels above the average are turned white. This initial approach yielded relatively effective results for images with moderate contrast (levels 1, 3, and 4), but it struggled to account for significant brightness variations and noise across the image.

Recognizing these limitations, we extended our research and identified a new method: Otsu's algorithm. Otsu's method calculates a global threshold value based on image characteristics such as overall brightness. This algorithm first constructs a histogram of pixel brightness levels, then calculates a variance across the brightness values. The threshold applied to the image is the one corresponding to the maximum variance in the histogram, effectively separating the foreground from the background in images with fairly uniform lighting. Although this method was far more effective than our initial approach, it had a key limitation: it applies a single brightness threshold across the entire image.

This meant that darker areas were processed in the same way as brighter areas, which could reduce binarization accuracy in images with complex lighting. To address this issue, we explored Sauvola's algorithm, an 'adaptive thresholding' method that adjusts the threshold dynamically based on local brightness levels. Unlike Otsu's algorithm, which uses a single threshold, Sauvola's approach calculates a unique threshold for each pixel. It considers both the local average brightness and the standard deviation around each pixel, ensuring a more context-sensitive binarization. This algorithm requires three parameters: two constants, k and R , which are used in calculating the adaptive threshold, and the window size, which defines the neighborhood of pixels around each target pixel. Sauvola's method is particularly effective because it calculates a threshold specific to each pixel's local environment, allowing for accurate binarization even in images with high variability in brightness.

However, a significant drawback arose with this approach: calculating the

mean, variance, and standard deviation for every pixel within a 25x25 pixel window proved to be computationally intensive and time-consuming. To mitigate this, we implemented integral images as an optimization. An integral image is a matrix of the same dimensions as the original image, where the value at any given position (x, y) represents the cumulative brightness of the pixel at (x, y) in the original image, plus the brightness values of all pixels above and to the left of it. This cumulative approach allows us to quickly compute the sum of brightness values within any subregion of the image, enabling faster calculations for mean and standard deviation.

For instance, if we denote $v(x,y)$ as the brightness value of the pixel at position (x, y) , then the value at position $(1, 1)$ in the integral image would equal $v(0,0)+v(1,0)+v(0,1)+v(1,1)$.

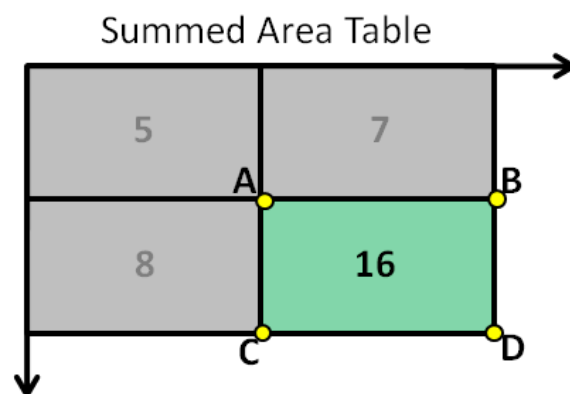
4	1	2	2
0	4	1	3
3	1	0	4
2	1	3	2

4	5	7	9
4	9	12	17
7	13	16	25
9	16	22	33

In our algorithm, we use two integral images: a standard integral image and a squared integral image, which contains the sum of the squares of pixel brightness values. These integral images enable a significant acceleration of the algorithm's calculations. For example, in a 25x25 pixel window, the previous version of the algorithm required 625 additions per pixel to compute the mean and another 625 additions to calculate the variance. This means that for a 512x512 pixel image, the algorithm had to perform over 150 million additions.

With the use of integral images, however, the number of additions per pixel

is reduced to just 4, regardless of the chosen window size. This reduction is achieved through an efficient cumulative summation technique, which allows us to calculate the sum of any rectangular region in constant time. For instance, in the illustration below, the sum of the green-highlighted area is obtained using the formula $D+A-C-B$, where each letter represents the cumulative sum at each corner of the region. This optimization allows our algorithm to perform complex brightness calculations swiftly, making it scalable and much more efficient for larger images.



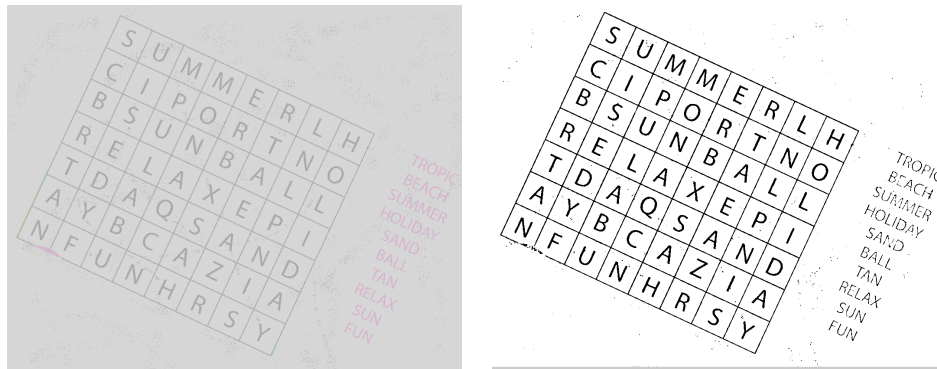
Thanks to this optimization, our algorithm's execution time was reduced from several seconds to approximately one-tenth of that time. The Sauvola algorithm is highly effective in accounting for the local characteristics of pixels within an image, but it performs less well in two specific cases: when the image contains significant noise and when the image has very low contrast.

To address the noise issue, we apply a Gaussian blur algorithm. Gaussian blur performs a weighted summation within a window of adjustable size, with a larger window producing a stronger blurring effect. Since we aim to preserve the original quality of the image as much as possible, we chose a sigma parameter of three, resulting in a 3x3 window for the weighted sum. This mild blur helps to even out image noise, making it easier for the Sauvola algorithm

to eliminate it.

As for the contrast issue, we developed an algorithm that examines and adjusts a pixel's value based on the local average brightness. Naturally, we use integral images to speed up these average calculations. The contrast adjustment algorithm takes two constants: a contrast factor (the higher the factor, the greater the contrast), which we set to 6.5, and the window size for calculating the local average, which we set to 35x35 pixels. The formula we use is as follows: $(v(x,y) - \text{mean}) * \text{factor} + \text{mean}$, where $v(x,y)$ is the pixel's brightness value.

In summary, our image preprocessing involves applying a Gaussian blur to reduce potential noise, converting the image to grayscale, adjusting the contrast, and finally applying the Sauvola algorithm to transform the image into black and white. Below are some examples of our results.

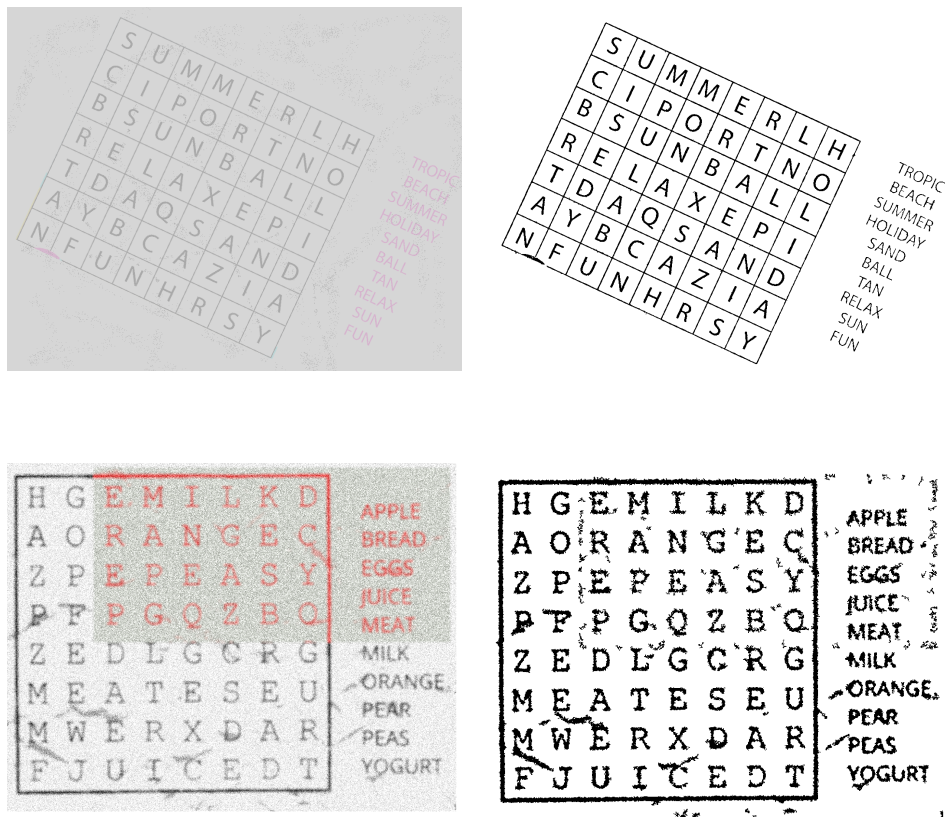


M	S	W	A	T	E	R	M	E	L	O	N	APPLE
Y	T	B	N	E	P	E	W	R	M	A	E	LEMON
R	R	L	W	P	A	P	A	Y	A	N	A	BANANA
R	A	N	L	E	M	O	N	A	N	E	P	LIME
E	W	L	E	A	P	R	I	A	B	P	R	ORANGE
B	B	I	L	B	B	W	B	R	L	A	Y	WATERMELON
K	E	M	P	M	A	W	L	R	A	R	B	GRAPE
C	R	E	P	R	N	R	E	R	R	G	R	KIWI
A	R	Y	A	Y	A	O	A	N	L	A	M	STRAWBERRY
L	Y	Y	A	R	N	E	R	K	I	W	I	PAPAYA
B	E	B	A	A	A	N	A	A	P	R	T	BLUEBERRY
Y	R	R	E	B	P	S	A	R	N	N	W	BLACKBERRY
Y	R	R	E	B	E	U	L	B	L	G	I	RASPBERRY
T	Y	P	A	T	E	A	E	P	A	C	E	

This method is quite effective; however, it still presents challenges: in images with significant noise, some noise persists even after applying our algorithm. To address this issue, we have implemented a post-processing algorithm aimed at reducing this remaining noise. The algorithm is straightforward: it iterates through all the 'blocks' of black pixels and calculates their sizes. Any blocks with a size smaller than 40 pixels are likely to be noise and are therefore removed by the algorithm.

This post-processing step also allows us to adjust the earlier parameters to enhance contrast more aggressively while keeping noise levels in check. For the Sauvola algorithm, we have opted for the following parameters: the constant k is now set to 0.6, R is set to 128, and the window size is reduced to 25 pixels. For contrast enhancement, we apply a factor of 9.35, maintaining the window size at 35x35 pixels.

The addition of this post-processing algorithm has yielded the following results, demonstrating an improvement in the overall clarity and quality of the images, while effectively mitigating residual noise.



Another problem with this binarization technique is due to the Gaussian blur that we apply. Indeed, by blurring the image, we inevitably lose precision. As a result, for some images, the letters composing the words to find get merged, making the character recognition much harder. However, we cannot simply remove the Gaussian blur from the algorithm as the binarization would let a large majority of the noise through, making the character recognition even harder. Thus, we decided to keep the algorithm as is in order to have the cleanest result possible since some images were not affected by this problem.

4 Additional tasks

4.1 Character Generator

We developed an application specifically designed to generate training images for a neural network. The purpose of this app was to provide a systematic way of creating large datasets of images, which are essential for training machine learning models, especially neural networks. Rather than manually collecting or curating images, which can be time-consuming and limited, the app automates the process to generate images on demand.

The training images were generated based on certain parameters and predefined rules. These parameters could include image dimensions, which characters to include, colors, rotations, or even warping the character itself. This variability in the generated images is crucial for training a robust neural network, as it allows the model to learn from a wide range of scenarios, which improves its generalization and performance on unseen data.

The app could create thousands of images quickly, allowing us to build customized datasets tailored specifically to the needs of the neural network. For example, we could generate images with different backgrounds, orientations, or even add noise and distortions to simulate the conditions found in our processed images, helping the neural network become more resilient to various challenges.

We focused on efficiency in the app's design to ensure it could handle large volumes of image generation without slowing down or crashing. In summary, this app proved to be an essential tool in creating the large-scale, varied datasets required for training our neural network, enabling more effective machine learning model development.

4.2 Website

In addition to developing the app for generating training images, we have also built a website to complement and enhance the overall functionality of the project. The website serves as a central hub where users can not only learn about the application but also easily access and acquire a copy of the software.

Our goal was to create an online presence that simplifies the process for users to find the information they need and quickly get started with the app. The website provides detailed documentation on how the app works, its features, and the steps for generating training datasets. We also included tutorials and use case examples, helping both novice and experienced users understand how to leverage the app for their specific needs.

In addition to informational content, the website also serves as a distribution platform, where users can download the software, ensuring they have quick and direct access to the latest version. We made the download process simple, with clear instructions and links to different versions or operating system compatibility options.

To further engage users, the website includes FAQs, a contact page, and possibly a community forum or discussion board, encouraging users to share their experiences, ask questions, or troubleshoot issues together. By building a website, we not only increased the visibility of the app but also established a centralized resource for ongoing support and updates.

Overall, the website acts as a comprehensive and user-friendly gateway to the app, ensuring that users can easily access the software, understand how to use it, and stay up to date with the latest improvements.

4.3 Word Search Generator

In addition to serving as a hub for the training image generation app, the website also houses our word search generator. This tool is another feature we integrated into the site, providing users with an interactive and engaging way to create custom word search puzzles.

The word search generator allows users to input their own list of words, select grid sizes, and customize other puzzle settings like what orientations words are allowed to be placed in, or whether the generator should prefer generating puzzles with more overlapping words. Once the user configures their puzzle, they can instantly generate a word search grid that can be printed, saved, or shared. This added functionality makes the website more versatile, appealing to a wider range of users who might be interested in fun, educational, or recreational activities.

By hosting the word search generator on the website, we provide an easily accessible tool that doesn't require users to download or install anything. The generator is entirely web-based, ensuring that it's available on any device with an internet connection, making it accessible to a broad audience.

Incorporating this feature into the website also helps attract a broader user base, as it expands the site's offerings beyond just the training image generation app. Whether users are looking for tools to help with learning, passing time, or simply having fun, the word search generator adds another layer of value and makes the site more engaging.

The word search puzzle generator works by first creating an empty array that represents the grid of the word search table, containing all the rows and columns that will eventually form the puzzle. This array serves as the foundation for placing the words into the puzzle.

To fill the grid, the generator places words one at a time, carefully choosing available spaces in the grid for each word. The placement of each word is determined by ensuring that it fits within the grid, either horizontally, vertically, or diagonally. The generator tries to place each word in the most suitable location, ensuring that the word doesn't overlap with other words already placed in the puzzle unless it shares common letters.

During the word placement process, the algorithm attempts to place each word a few different times in different positions within the grid. This trial and error approach ensures that the word fits properly. If the word cannot be placed in the grid after several attempts, the generator assumes that it is not possible to fit the word in and will backtrack or remove previous words to make room for new ones.

Once all the words are placed, the remaining empty spaces in the grid are randomly filled with letters, ensuring that they don't inadvertently form new words. This creates a completed puzzle where users can search for the listed words.

If at any point during the word placement process the generator encounters difficulty and cannot find a suitable spot for a word after a reasonable number of attempts, it will give up and notify the user that it's not possible to create a valid puzzle with the provided words and grid size. The user may then adjust the word list, grid size, or difficulty and try again.

This algorithm ensures that the puzzles are solvable, and by utilizing a mix of trial, error, and backtracking, the generator creates challenging yet fair word search puzzles every time.

5 Conclusion

We believe we have have progressed well throughout the project. Even though we did not achieve all of our goals, we remain satisfied with the results we were able to obtain considering the difficulties we encountered and the work charge we had outside of the project. Thanks to the OCR project, every member of the team acquired new valuable knowledge that will be useful for our future careers.