From Vision to Typing: AI Automation for Mastering Typing Games

Naris Chanpaiboonkit[a,*]1, Asst. Prof. Ekarat Rattagan, Ph.D.[b]

Graduate School of Applied Statistics,
National Institute of Development Administration,
148 Serithai Road, Klong-Chan, Bangkapi, Bangkok Thailand 10240

Abstract

Typing-based games are traditionally designed to enhance human typing skills, but they also provide a structured, high-speed environment for testing AI-driven text recognition and automation. This study explores the application of deep learning to develop an AI system capable of autonomously playing these games by recognizing and inputting text in real time. The objective extends beyond achieving high in-game performance to evaluating AI methodologies for dynamic text-based interactions. Five AI-driven approaches were benchmarked, integrating YOLOv11 for object detection, EasyOCR and Tesseract for text recognition, and SymSpell for error correction, with additional functionalities such as object tracking and fallback mechanisms. The best-performing approach, YOLOv11 + EasyOCR + SymSpell, achieved the highest throughput (18.778 FPS) while maintaining high accuracy and efficiency, significantly outperforming the baseline brute force method. The results highlight the effectiveness of combining object detection, OCR, and real-time correction in automated text processing. Beyond gaming, this research provides insights into AI applications requiring real-time text recognition and interaction in dynamic environments, such as automated license plate recognition (ALPR), quality assurance (QA) in user interface automation, and AI-driven task automation in industrial and software applications. These findings emphasize the importance of integrating OCR, object detection, and intelligent input automation for enhancing AI-driven interactions across multiple domains.

1. Introduction

Typing-based games have long been utilized as interactive tools to enhance human typing skills. While primarily designed for learning, these games also provide a structured, high-speed environment that can serve as a testbed for AI-driven automation in real-time text processing. Unlike controlled laboratory conditions, typing-based games present dynamic, time-sensitive challenges that require AI systems to accurately recognize, interpret, and input text under strict constraints. The ability to perform real-time text recognition and automated input is critical for applications beyond gaming, including automated document processing, intelligent task automation, and AI-driven user interface interaction. Several studies have explored real-time OCR and object detection in dynamic environments. However, existing research has not fully addressed the challenges of integrating these techniques into high-speed text interaction scenarios. Despite advancements in OCR and deep learning-based automation, many systems still struggle with noisy or rapidly changing text environments and the trade-offs between speed and precision in high-speed automation tasks. To address these challenges, this research presents an AI-driven real-time text recognition and automation framework that integrates multiple deep learning-based techniques, including real-time text region detection, OCR-based text recognition, error correction, and object tracking for stability across frames in fast-moving scenarios. The proposed system is evaluated across various game environments, comparing multiple AI-driven methods for text recognition and automated input. The results demonstrate significant improvements in detection accuracy, processing efficiency, and overall gameplay performance compared to baseline approaches. Beyond gaming, this study contributes to AI-driven real-time automation by developing a robust text recognition system capable of handling high-speed and dynamic input environments. It further highlights the potential of typing-based games as an AI testbed, with broader applications in automated license plate recognition (ALPR), AI-driven UI automation, and OCR-based quality assurance (QA) systems. By addressing the limitations of OCR and AI-driven automation in dynamic settings, this research lays the foundation for broader real-time AI applications that require high-speed text recognition and interaction.

* Corresponding author.
E-mail address: 6520422025@stu.nida.ac.th

2. Literature Review

The integration of object detection, OCR, and AI-driven automation has been extensively studied for applications requiring real-time text recognition and interaction. Several studies have demonstrated the effectiveness of deep learning-based models such as YOLO, Tesseract, and EasyOCR in enhancing recognition accuracy and efficiency in dynamic environments. Thai et al. [1] introduced PLayerTV, an AI-driven player tracking system for soccer highlight generation, showcasing how real-time object detection can be applied in fast-moving environments. This aligns with our approach, where YOLOv11 is employed to detect and track text-based elements in a dynamic game setting under strict time constraints. Similarly, Johnson et al. [2] explored video games as a training platform for computer vision models, reinforcing the idea that games can serve as an effective AI testbed, much like our application in automated text input for typing-based games. The combination of OCR and YOLO-based object detection has been widely explored for automated text recognition. Prajapati et al. [3] and Apriyadi & Antoni [5] demonstrated that integrating YOLO and Tesseract OCR significantly enhances license plate recognition (ALPR), proving the effectiveness of deep learning-based text detection in real-world, high-speed scenarios. Likewise, Kim [4] developed Easy-Yolo-OCR, a system that integrates YOLO with EasyOCR for real-time text extraction, which directly supports our methodology of combining YOLOv11 with EasyOCR and Tesseract for game-based text recognition. Optimizing OCR models for different contexts has also been a major research focus. Ranathunga et al. [9] adapted Tesseract OCR for Tamil and Sinhala fonts, emphasizing the need for fine-tuning OCR models for non-standard text styles, a concept relevant to our game-based text recognition task. Vedhaviyassh et al. [14] conducted a comparative analysis of EasyOCR and TesseractOCR, concluding that EasyOCR provides higher accuracy for structured text recognition, influencing our decision to integrate both OCR engines for optimal performance. Additionally, Smith [13] provided a comprehensive overview of Tesseract OCR, highlighting its adaptability for real-world text recognition tasks. Preprocessing and error correction play a critical role in improving OCR accuracy. Pandey et al. [7] compared image preprocessing techniques, demonstrating how denoising and contrast enhancement improve recognition accuracy—an approach we incorporate in text detection refinement before OCR processing. Furthermore, Garbe [12] introduced SymSpell, an efficient spell-checking and error correction algorithm that reduces OCR errors, which is integrated into our system to enhance real-time text recognition accuracy. Advancements in deep learning-based AI automation have further influenced our approach. Kumar et al. [6] demonstrated intelligent document parsing using OCR and AI, which parallels our objective of automating text input through AI-driven recognition and automation. Meanwhile, Ge et al. [8] presented a detailed overview of YOLOv11's architectural improvements, reinforcing our choice of YOLOv11 for real-time object detection in gaming environments. Real-time object tracking is another essential component in our study. Bewley et al. [10] introduced SORT (Simple Online and Realtime Tracking), an efficient algorithm that we adopt to track Textbox positions and game elements across frames. Additionally, Du et al. [15] developed PP-OCR, a lightweight OCR system optimized for real-time applications, providing insights into how OCR models can be streamlined for faster processing, a crucial factor in our AI automation framework. These studies form the foundation for integrating object detection, OCR, and real-time tracking technologies to enhance automated text recognition in dynamic environments. However, most existing research focuses on static text extraction or document processing, whereas our study applies these technologies in a high-speed, game-based setting. By combining YOLOv11 for detection, OCR for recognition, SymSpell for error correction, and SORT for tracking, our framework aims to improve both accuracy and efficiency in real-time AI-driven text automation.

Table 1 Summary of Studies Investigating Object Detection and OCR Applications

Year	First Author	Task	Model Training Methodology	Metric
2007	Smith [13]	OCR Engine	Tesseract OCR	Recognition Accuracy, Processing Speed
2021	Seren [1]	Object Tracking	SORT (Simple Online and Real-Time Tracking), OpenCV	Precision, Recall, mAP
2023	Bellotti [4]	Object Detection + OCR	YOLOv4, Tesseract	Precision, Recall, F1 Score
2022	Kurniawan [9]	Preprocessing for OCR	Linear Models, Nonlinear Models	Accuracy Improvement
2022	Vedhaviyassh [14]	License Plate Recognition	YOLOv5, EasyOCR, TesseractOCR	Recognition Accuracy, Comparative Performance
2023	Mohammad Abdullah [6]	OCR Adaptation	Tesseract OCR	Accuracy, Flexibility
2024	Agrawal [10]	Object Detection	YOLOv11	Precision, Recall, Scalability

3. Solution Idea

In this project, we developed an AI-driven framework to autonomously play a typing-based game at its highest difficulty level. This framework leverages advanced computer vision and deep learning methodologies to tackle the dual objectives of accurately and efficiently typing game-specific text (primary tasks) and identifying and collecting map-specific items (secondary tasks). By employing YOLOv11 for real-time object detection and EasyOCR for text recognition, the system achieves rapid and accurate text processing, while SymSpell-based error correction ensures improved typing precision. Moreover, SORT (Simple Online and Realtime Tracking) was integrated to maintain contextual tracking of objects across frames during high-speed gameplay.

The process involves preprocessing, object detection, text recognition, and error correction before automated typing. Additionally, five distinct methods were benchmarked, each combining various techniques like brute force and OCR models (EasyOCR, Tesseract), to evaluate their performance based on metrics such as typing accuracy, score, completion time, FPS.

3.1. Performance Metrics

The system's performance was evaluated using metrics that reflect its effectiveness in gameplay automation. These include Score Avg, Accuracy (Acc Avg), Letters Typed Avg, Perfect Words Avg, Kills Avg, Best Combo Avg, and Goregasm Avg. All these metrics are provided by the game itself as part of its built-in evaluation system, which summarizes the performance at the end of each level. These metrics allow us to assess the system’s ability to type accurately, maintain high-performance streaks, and handle game challenges effectively. In contrast, the FPS (Frames Per Second) metric was independently assessed by the researchers during the experiments. FPS reflects the computational performance of the system in real time, indicating how efficiently the AI processes frames during gameplay. This was measured using latency data collected during frame processing and converted into FPS values. While the in-game metrics evaluate the system's gameplay success, the FPS metric provides insight into the system's technical efficiency and real-time capability. Together, these metrics offer a comprehensive evaluation of both the gameplay effectiveness and technical performance of the AI system.

3.1.1 Accuracy (Acc Avg):

Accuracy measures the proportion of correctly typed characters compared to the total number of characters typed. It is calculated by dynamically evaluating the number of correct inputs against the total inputs during gameplay.

As each character is typed, the accuracy metric updates in real time, providing an immediate reflection of the system's precision. This dynamic approach ensures that accuracy is consistently representative of performance throughout the gameplay session.

Accuracy serves as a vital indicator of the system's effectiveness, offering a clear metric for assessing its ability to type words correctly with minimal errors. A higher accuracy rate underscores the system's proficiency in maintaining precision, directly contributing to better gameplay outcomes and higher scores. This metric not only evaluates the technical capabilities of the system but also highlights areas for potential improvement in text recognition and input processes.

$$Accuracy\ (Acc\ Avg) = \frac{Number\ of\ Correctly\ Typed\ Characters}{Total\ Number\ of\ Characters\ Typed} \times 100 \quad (1)$$

3.1.2 Score Avg:

In *The Typing of the Dead: Overkill*, the scoring system is intricately designed to reward players for precision, efficiency, and strategic decision-making during gameplay. Scores are primarily influenced by the player's typing accuracy and the ability to maintain error-free streaks. Typing with high accuracy ensures that players earn higher scores, as consistent precision often leads to combo bonuses and Perfect Words Bonuses, which significantly enhance overall performance.

The combo meter plays a vital role in the scoring mechanism. By typing without errors and maintaining uninterrupted streaks of successful inputs, players build up their combo meter. This multiplier increases the points awarded for each correct word typed consecutively, encouraging players to focus on accuracy and rhythm in their typing.

Additionally, players can earn bonus points by completing specific challenges within the game, such as eliminating a designated number of enemies within a time limit or completing a level without taking damage. These objectives test the player's ability to type accurately and quickly under pressure, further adding depth to the scoring system.

Rescuing civilians during gameplay provides another opportunity to earn additional points. This feature demands quick and precise typing, as players must prioritize their targets effectively to protect the civilians and secure the bonuses associated with these actions.

Overall, the scoring system is designed to motivate players to improve their typing skills by rewarding accuracy, speed, and consistency. By aligning the gameplay objectives with these skills, the game not only enhances the player's proficiency but also provides a satisfying and engaging gaming experience.

3.1.3 Letters Typed Avg:

Letters Typed Avg represents the average number of characters typed during gameplay, reflecting typing efficiency. A lower average indicates fewer unnecessary keystrokes, showcasing the system's precision and ability to input only the required characters, essential for smooth and effective gameplay.

3.1.4 Perfect Words Avg:

Perfect Words Avg refers to the average number of consecutive words typed without errors. This metric is essential as it demonstrates the system's ability to sustain high accuracy over extended sequences, reflecting its reliability in maintaining error-free performance. Consistently achieving high values in this metric indicates an advanced level of precision and efficiency in gameplay typing tasks.

3.1.5 Goregasm Avg:

Goregasm Avg captures the average duration of the game's highest performance streaks. This metric evaluates the system's capability to perform at peak levels continuously. Sustained high performance, as indicated by a high Goregasm Avg, highlights the robustness and precision of the typing system under demanding gameplay conditions.

3.1.6 Player deaths Avg:

Player Deaths Avg represents the average number of player deaths recorded during gameplay. This metric is a direct indicator of the system's robustness and its ability to avoid failure conditions that could disrupt gameplay. A low Player Deaths Avg signifies a reliable and effective system capable of maintaining player survivability throughout the game.

3.1.7 Best Combo Avg:

Best Combo Avg measures the average length of the game's highest combo streaks, reflecting the ability to chain successful actions together without interruption. This metric is vital for assessing the system's capability to sustain high-performance sequences, contributing to higher overall scores and a seamless gameplay experience.

3.1.8 Kills Avg:

Kills Avg refers to the average number of enemies defeated during gameplay. This metric demonstrates the effectiveness of the system in handling combat scenarios, indicating its ability to interact with the game environment accurately and efficiently. High values in this metric showcase the system's capability to execute gameplay actions with precision and speed.

3.1.9 Time Avg:

Time Avg represents the average duration required to complete a level within the game. This metric serves as a measure of the system's efficiency in achieving gameplay objectives. A lower Time Avg indicates the system's capability to perform tasks swiftly while maintaining accuracy and precision, highlighting its effectiveness in navigating and completing game challenges within minimal time. Conversely, higher values may suggest inefficiencies or delays in recognizing and responding to game inputs, underscoring areas for improvement.

3.1.10 Throughput (FPS):

Throughput, measured in Frames Per Second (FPS), represents the system's ability to process frames efficiently during gameplay. FPS indicates how many frames the system can process in one second, reflecting its computational speed and real-time capability. A higher FPS value implies faster and more efficient processing, which is critical for real-time applications like gameplay automation. The FPS was calculated dynamically using the latency per frame during the detection process. For each frame, the latency (time taken to process the frame) was recorded in milliseconds and converted to FPS as follows:

$$FPS = \frac{1}{Latency\ (second\ per\ frame)} \tag{2}$$

3.2 Methods:

Five distinct methods were developed and benchmarked to compare performance metrics, as outlined below:

3.2.1 Brute Force Method:

The brute force approach operates by systematically typing all possible characters in sequence until the correct word is identified. The process begins by iterating through lowercase letters (a-z), followed by numbers (0-9), and then special characters. For each position in the target word, the system types one character at a time and checks whether it matches the desired input. If the correct character is identified, the process moves to the next position, repeating this cycle until the entire word is completed. This method ensures that every possible combination of characters is tested, guaranteeing that the correct word will eventually be entered. However, the process does not rely on any contextual understanding or prediction of the text, making it purely exhaustive and deterministic.

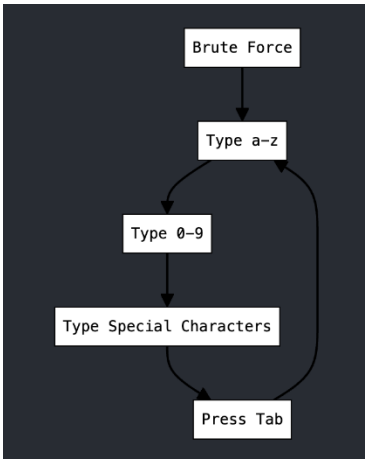


Figure 1 illustrates the brute force method

3.2.2 YOLOv11 + EasyOCR:

This method combines YOLOv11 for detecting text regions in the game environment and EasyOCR for recognizing the detected text. The process begins with YOLOv11, which identifies and classifies bounding boxes containing text elements or other objects. These bounding boxes are categorized into different classes such as Text Classes and Collectible Classes, allowing the system to perform class-specific actions effectively. For Text Classes, the system extracts the bounding box, crops the relevant region, and processes the cropped image through EasyOCR to interpret the text. The cropping process ensures that only the detected text is passed to the OCR engine, reducing noise and improving recognition accuracy. Once the text is successfully recognized, it is converted into input commands and typed into the game in real-time. For Collectible Classes, the system performs predefined actions such as pressing the "Tab" key to interact with objects in the game environment that are not related to text input. This separation of tasks allows the system to handle text-based and non-text-based interactions efficiently. The seamless integration of YOLOv11 and EasyOCR allows the system to dynamically adapt to various text recognition tasks, enabling real-time processing of game elements. By focusing on extracting and recognizing text accurately, this method provides a robust pipeline from object detection to text input, ensuring smooth and effective gameplay automation.

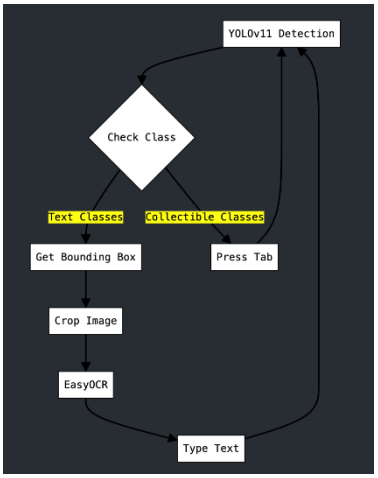


Figure 2 illustrates the YOLOv11 + EasyOCR workflow.

3.2.3 YOLOv11 + Tesseract OCR:

This approach integrates YOLOv11 for detecting text regions and Tesseract OCR for recognizing the detected text. The process begins with YOLOv11 identifying bounding boxes around text elements or other objects in the game environment. For text classes, the system extracts the bounding box, crops the corresponding region, and converts it into grayscale to enhance recognition quality. The cropped image is then processed using Tesseract OCR, which interprets the text within the region and outputs the recognized characters. The recognized text is subsequently cleaned and formatted before being typed as input into the game. For non-text classes, such as collectibles, the system

performs specific actions, such as pressing the "Tab" key to interact with the object. This pipeline provides a straightforward implementation of object detection and text recognition, leveraging the capabilities of Tesseract OCR to interpret text from the cropped regions.

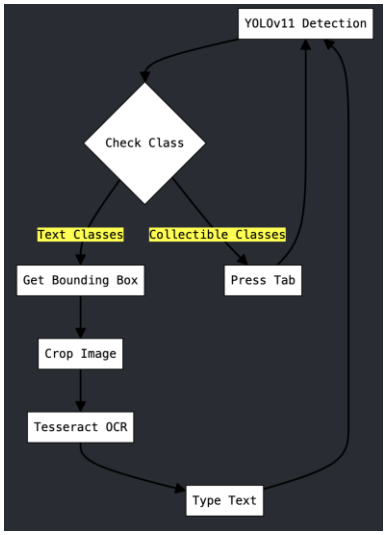


Figure 3 illustrates the YOLOv11 + Tesseract OCR orkflow

3.2.4 YOLOv11 + EasyOCR + SymSpell:

This method builds upon the YOLOv11 + EasyOCR pipeline with the addition of SymSpell for text correction, specifically in handling Textbox classes. Similar to the YOLOv11 + EasyOCR approach, YOLOv11 detects objects in the game environment and classifies them into categories such as Text Classes and Collectible Classes. For Collectible Classes, predefined actions like pressing the "Tab" key are executed to interact with non-text game elements. For Text Classes, the bounding boxes are cropped and processed into grayscale images before being passed to EasyOCR for text recognition. The recognized text is then sent through SymSpell, which performs error correction by comparing the text to a preloaded frequency dictionary. This helps identify and correct potential errors in the OCR results, improving the accuracy of the text input. The corrected text is then typed into the game environment as input. By integrating SymSpell into the pipeline, this method enhances the system's ability to handle errors in text recognition, particularly for complex or low-quality text in Textbox classes, while maintaining the efficiency of the YOLOv11 + EasyOCR pipeline for other types of objects.

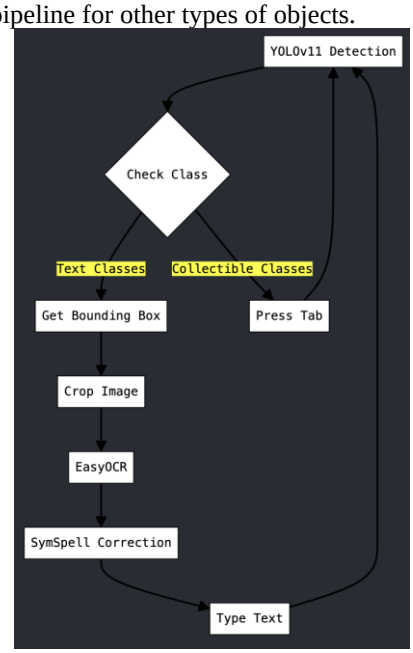


Figure 4 illustrates the YOLOv11 + EasyOCR + SymSpell workflow

3.2.5 YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force:

This method integrates YOLOv11 for text detection, EasyOCR for recognition, SymSpell for error correction, SORT for object tracking, and brute force as a fallback. After YOLOv11 detects objects, the system classifies them into three categories: Quick Text, Other, and Textbox. For Quick Text, EasyOCR extracts and types the recognized text immediately. For Other objects, predefined actions such as pressing the "Tab" key are executed. Textbox objects require additional complexity, where SORT tracks each textbox ID across frames to ensure the text is correctly processed. EasyOCR first attempts to recognize and input the text within a set number of frames. If the textbox persists beyond this limit, SymSpell corrects potential recognition errors, and the corrected text is retried. Should the issue remain unresolved, brute force typing is used as a final fallback to clear the textbox and maintain progress. This approach ensures robust handling of text-based challenges, combining tracking, recognition, and fallback mechanisms to maintain accuracy and efficiency during gameplay.

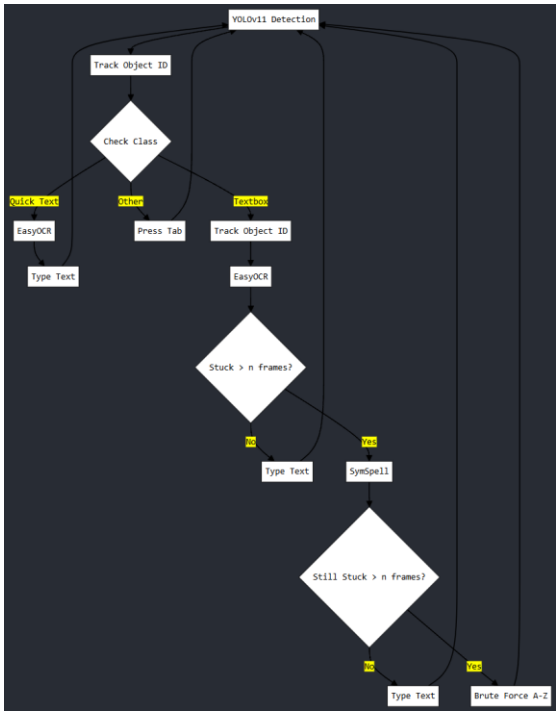


Figure 5 illustrates the YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force workflow

4. Experimental Setup

Figure 6 Sample frames from The Typing of the Dead: Overkill game showing various detected objects: textboxes for words, quick text for single characters, and collectible items (health, comic, poster, record, and figure). Each frame demonstrates the YOLO model's ability to identify and classify different game elements accurately.

4.1 Dataset

The dataset used in this project was generated from gameplay sessions in The Typing of the Dead: Overkill, a fast-paced, typing-based action-horror game developed by SEGA and released on Steam. Unlike traditional first-person shooter (FPS) games where players manually control movement and actions, this game follows an on-rails shooter format, meaning the player's character moves automatically through the game world, encountering enemies and environmental challenges along the way. The player's primary responsibility is to type words or phrases displayed on the screen as quickly and accurately as possible to eliminate enemies before they attack. In addition to enemy encounters, players must also respond to interactive elements, including quick-time events and collectible objectives, which influence gameplay progression and overall scoring. These mechanics make The Typing of the Dead: Overkill an ideal testbed for real-time text recognition and automated input systems, as it requires rapid and precise text processing under dynamic conditions.

To create a diverse and representative dataset, images were captured from three difficulty levels (Easy, Medium, and Hard) and three selected stages from the total nine available in the game. A total of 360 images were collected and divided into 149 images for the training set and 66 images for the test set. These images were designed to capture both primary and secondary in-game tasks. The primary task involves detecting text-based elements, such as textboxes and quick text prompts, which are essential for defeating enemies and progressing through levels. The secondary task focuses on identifying hidden collectible items, including comics, figures, posters, and records, which contribute to bonus scores and influence the player's final performance rating. Figure 6 presents sample frames from the game, illustrating the detected objects, including textboxes for words, quick text prompts for single-character inputs, and collectible items such as health, comic, poster, record, and figure. The YOLO model accurately identifies and classifies these game elements, which are fundamental for developing an AI-driven automation framework.

To facilitate object detection, the collected images were processed and annotated using Roboflow, a cloud-based annotation tool. Eight distinct object classes were labeled in the dataset to enable real-time AI-based recognition. Textboxes contained words that players needed to type to defeat enemies, while quick text prompts consisted of single-character inputs that required immediate action to avoid damage. Health and slow items represented gameplay-enhancing elements, where health items restored player vitality and slow items temporarily reduced the speed of enemy attacks, allowing players more time to react. The remaining categories—comic, figure, poster, and record—were collectibles scattered throughout the game that contributed to secondary objectives and increased the final game score.

To enhance the robustness and generalization of the detection model, the dataset underwent preprocessing and augmentation using Roboflow's integrated pipeline. These preprocessing steps followed best practices in data

augmentation for deep learning-based OCR and object detection. To ensure uniformity in input data, all images were resized to a consistent resolution of 640×640 pixels before augmentation. Several augmentation techniques were applied to increase dataset variability and improve model adaptability to different in-game conditions. These included rotation within a range of -12° to +12° to simulate different viewing angles, grayscale conversion on 15% of the images to account for variations in lighting conditions, and random cropping or zooming with a zoom range of 0% to 21% to introduce size variation in detected objects while preserving their semantic content. These augmentation strategies ensured that the model would be resilient to changes in game conditions, such as different lighting effects, motion blur, and variations in text positioning.

After augmentation, the dataset was split into training and validation sets, comprising 94% and 6% of the total augmented images, respectively. This resulted in 912 training images and 56 validation images. Each image was stored in .png format, accompanied by a corresponding .txt file containing class and bounding box annotations in YOLO format. These labels specified the normalized x and y center positions, bounding box width and height, and the class index corresponding to the detected object. This structured dataset preparation process ensured that the YOLOv11 model could effectively detect and classify objects within the fast-paced, dynamic gaming environment while maintaining high accuracy under varying conditions.

4.2 Exploratory Data Analysis (EDA)

The dataset used for this project underwent an extensive exploratory data analysis (EDA) phase to understand its characteristics and ensure its suitability for training the object detection and OCR models. The dataset contains a total of 360 images, which are divided into training and validation sets. The dataset includes annotations for eight distinct classes: textbox, quick_text, slow, health, figure, record, comic, and poster. The average image size is 1.23 megapixels, with a median aspect ratio of 1475x831 pixels.

The annotations within the dataset total 687 across all images, averaging 1.9 annotations per image. These annotations represent objects of interest in the game environment. The class distribution indicates that the majority of the annotations belong to the textbox class (456 annotations), followed by quick_text (58 annotations) and slow (57 annotations). Other classes, such as figure, record, comic, and poster, have fewer annotations, reflecting their rarity in the dataset.

The dataset also includes a histogram of object counts per image. The majority of images (185) contain 2–3 annotations, while 148 images contain a single annotation, and only 25 images contain between 4–5 annotations. This distribution highlights the variability in object density across images, providing a robust basis for training a model capable of handling diverse scenes.

To address the imbalance in annotations and enhance model generalizability, data augmentation was performed using the Roboflow platform. Preprocessing steps included resizing all images to 640x640 pixels and applying augmentation techniques such as cropping, rotation (between -12° and 12°), grayscale conversion (applied to 15% of the images), and zoom adjustments (0–21%). The augmented dataset ensures that the models are exposed to a wide variety of scenarios, improving their robustness in detecting objects in real-world gaming environments.

Additionally, the dataset analytics reveal that there is one missing annotation and one null example. This anomaly was handled during the preprocessing phase to ensure that only valid and properly annotated data were included in the training process.

By leveraging insights from the EDA phase, the dataset was optimized for training, balancing the class distribution and ensuring the model's ability to detect both primary and secondary objects in the gaming environment effectively.

4.3 YOLOv11 Training and Validation

Table 2 YOLOv11 training and validation configurations with key hyperparameters and settings.

Description	Value
Model Architecture	YOLOv11x
Epochs	300
Image Size	640 x 640
Patience	50
Optimizer	AdamW
Initial Learning Rate	0.01
Final Learning Rate	0.01
Warmup Epochs	5
Momentum	0.937
Weight Decay	0.0005
Box Loss Weight	2
Classification Loss Weight	1
Distribution Focal Loss Weight	1.5
Mixed Precision	Enabled (Half)
Data Augmentation - Hue	0.5
Data Augmentation - Saturation	0.7
Data Augmentation - Brightness	0.4
Validation Enabled	TRUE
Save Checkpoint Every	10 epochs
Single Class Training	FALSE
Plot Training Metrics	Enabled

According to [8], YOLOv11 introduces several significant architectural enhancements over its predecessors, incorporating improved feature extraction capabilities through a modified backbone network and enhanced cross-stage partial networks. These modifications contribute to better handling of multi-scale object detection, which is particularly beneficial for detecting varying sizes of text elements in our application. The model implements an advanced loss function system that combines multiple components: box loss for localization accuracy, classification loss for object categorization, and distribution focal loss for better handling of class imbalances [8]. Additionally, YOLOv11's improved anchor-free detection system and adaptive feature alignment enhance the model's ability to detect objects of varying scales and aspects ratios, which proved especially valuable in our implementation, where text elements appear in different sizes and orientations.

The training process of the YOLOv11 model demonstrated a consistent decline in all types of losses, including box loss, classification loss, and distribution focal loss. The smooth downward trends in the loss metrics, as illustrated in the training plots, reflect the model's effective optimization and convergence over 66 epochs. Early stopping was triggered as no significant improvement was observed beyond epoch 16. The best-performing model was saved at this point, with validation results showcasing strong overall performance.

Upon evaluating the model on the validation set, the results showed a mean Average Precision (mAP50) of 82.6% and an mAP50-95 of 57.4%. Precision and recall metrics varied significantly across classes. Notably, the textbox class achieved near-perfect precision (99.1%) and recall (100%), with an mAP50 of 99.5%. Similarly, the quick_text class also performed well, with precision and recall scores of 89.5% and 95.4%, respectively, and an mAP50 of 96.2%.

These strong results are attributed to the relatively higher number of instances for these classes, as they are the primary objectives in this dataset.

Conversely, other classes such as comic, figure, health, poster, and record exhibited lower precision and recall scores. For example, the figure class achieved an mAP50 of only 66.3%, while the record class had an mAP50 of 49.7%. These disparities can be traced back to the limited amount of training data available for these classes, resulting in the model's reduced ability to generalize effectively. The overall class imbalance in the dataset, particularly the dominance of textbox and quick_text annotations, is a critical factor influencing these results.

The precision-confidence curve further illustrates the variability in class-specific performance. While textbox and quick_text exhibit steep and consistent precision trends with increasing confidence thresholds, other classes show irregular trends, reflecting the challenges posed by limited data.

The confusion matrix highlights the model's high confidence and correct predictions for the textbox and quick_text classes. However, there is notable misclassification among less-represented classes such as comic, figure, and record. For instance, several instances of figure were misclassified as textbox, emphasizing the need for additional data or better augmentation strategies for minority classes.

In summary, while the YOLOv11 model demonstrates robust performance for primary classes like textbox and quick_text, the scarcity of data in other classes significantly impacts the overall performance. Addressing this imbalance through data augmentation or targeted collection of additional data for underrepresented classes could further enhance the model's accuracy across all categories.

4.4 Brute Force Method for Text Recognition and Typing

The brute force method is designed to attempt typing text by iterating through all characters in a predefined set (0-9, a-z, ', -) one by one for each position in the target word. This process operates without prior knowledge of whether the typed character is correct until the game accepts it and moves to the next position.

For example, if the target word is "I love you," the system starts by iterating through characters from a to - for the first position until it matches the correct character l. Once the correct character is identified, the game advances to the next position, but the system continues iterating through the remaining characters in the current set. Similarly, in the second position, the system identifies l in the first round of iteration, while in the third position, it finds o. In subsequent iterations, it identifies v, which is next to o in the character set. This process continues until all characters in the word are correctly typed.

Although the brute force method is straightforward and guarantees successful text entry in all cases, its drawbacks include an excessively high number of typed characters compared to other methods and significantly lower accuracy when compared to advanced analytical or algorithmic approaches. However, brute force excels in reliability and serves as a baseline for comparing more sophisticated and accurate methods later in the study.

4.5 OCR Framework Integration for In-Game Text Recognition

In this project, we experimented with two OCR frameworks, EasyOCR and Tesseract OCR, to evaluate the performance of each method in recognizing text within the gaming environment. The two frameworks were tested separately, not concurrently. EasyOCR was selected for its robust capability to recognize dynamic text within images [4], while Tesseract OCR excelled in recognizing structured text such as UI components and tooltips [3, 9].

Prior to feeding input into EasyOCR and Tesseract OCR, preprocessing steps were applied to enhance recognition efficiency. For the textbox class, grayscale conversion was performed to simplify the image and reduce unnecessary noise. For the quick_text class, additional preprocessing was applied, including grayscale conversion and cropping to eliminate peripheral noise and focus on the relevant text [7, 11].

For EasyOCR, the system was configured using the Reader class with the English language model, specified as Reader(['en'], gpu=torch.cuda.is_available()). This ensured GPU acceleration when available, which significantly improved inference speed during gameplay. On the other hand, Tesseract OCR was configured using the pytesseract.image_to_string function, with the language parameter explicitly set to English (lang='eng'). This configuration enabled consistent text recognition performance across all datasets.

The experiments were conducted independently using the same dataset for both EasyOCR and Tesseract OCR, allowing for a direct comparison of their performance in terms of accuracy and speed [3, 5]. The system was run on an NVIDIA RTX 4060 Ti GPU to ensure real-time processing during gameplay, enabling seamless integration into the gaming environment [2].

These results provided key insights into the strengths and limitations of EasyOCR and Tesseract OCR for recognizing text in gaming contexts, forming a foundation for improving OCR systems tailored to specific applications in the future [6].

4.6 Text Correction with SymSpell for OCR Outputs

In this project, we utilized SymSpell to correct errors arising from the OCR frameworks specifically for the textbox class. Texts in this class are often longer and more complex, making them more prone to recognition errors that require post-processing for improved accuracy. SymSpell is an algorithm designed for fast and efficient text correction by leveraging a precompiled dictionary for word lookup and corrections based on edit distance. This approach significantly enhances the accuracy of recognized text by addressing errors from the OCR process.

For the setup, we configured SymSpell with parameters tailored to the game's context. The maximum edit distance was set to 2, allowing minor recognition errors to be corrected without compromising computational efficiency. Additionally, the prefix length was set to 7 to optimize memory usage and accelerate the search process. A custom dictionary was prepared, incorporating game-specific vocabulary alongside general English terms, ensuring comprehensive error correction coverage.

Once the OCR frameworks recognized text from the textbox class, the resulting text was processed through SymSpell for error correction. This step involved fixing misspelled words or completing incomplete terms, thereby improving the quality of the recognized text. For example, a misrecognized word like "Gmae" could be corrected to "Game," enhancing the overall readability and utility of the output.

SymSpell was not applied to texts from the quick_text class, as these were typically single characters or short words that did not require further correction. Limiting SymSpell's application to the textbox class reduced unnecessary computational overhead and maintained the system's overall speed.

The integration of SymSpell into this project significantly improved the accuracy of text recognition for the textbox class, particularly in scenarios where OCR errors might otherwise hinder usability. By correcting errors efficiently *and effectively, SymSpell enhanced the reliability of the OCR pipeline, ensuring both accuracy and real-time performance suitable for the gaming environment.*

4.7 Experimental Methodology

The experimental evaluation involved testing the performance of five approaches: Brute Force Method, YOLOv11 + EasyOCR, YOLOv11 + Tesseract OCR, YOLOv11 + EasyOCR + SymSpell, and YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force. These methods were evaluated in the gaming environment to assess their effectiveness and efficiency in automated text recognition and input.

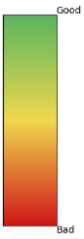
The experiments were conducted on levels 5 through 8 of the game at the highest difficulty setting. These levels were distinct from the initial levels used for training YOLOv11. During each experiment, the system operated autonomously, without human intervention, until the level was completed. At the end of each level, the game displayed the results, including scores and performance metrics as outlined in Section 4.1.

Each method was tested twice per level across four levels, resulting in a total of 10 trials per method. This approach ensured a consistent evaluation of all methods under similar conditions. The recorded metrics were subsequently analyzed to determine the strengths and limitations of each method in real-world gaming scenarios.

5. Experimental Result

Table 3 Comparative Evaluation of Average Gameplay Metrics Across All Maps for Each Method

Method	Brute Force	Yolo11+ EasyOCR	Yolo11+ EasyOCR+SymSpell +SortTrack+Brute Force	Yolo11+EasyOCR+SymSpell	Yolo11+Tesseract
Score Avg (points)	60,543.30	91,919.40	**102,806.90**	82,600.60	43,468.10
Acc Avg (%)	4.36	**84.53**	84.39	77.82	76.34
Letters Typed Avg (count)	**45,930.90**	2,640.20	2,676.30	2,526.80	2,943.10
Player Deaths Avg (count)	0.00	0.00	0.00	0.40	**2.00**
time Avg (seconds)	12.16	9.99	11.08	10.31	**12.28**



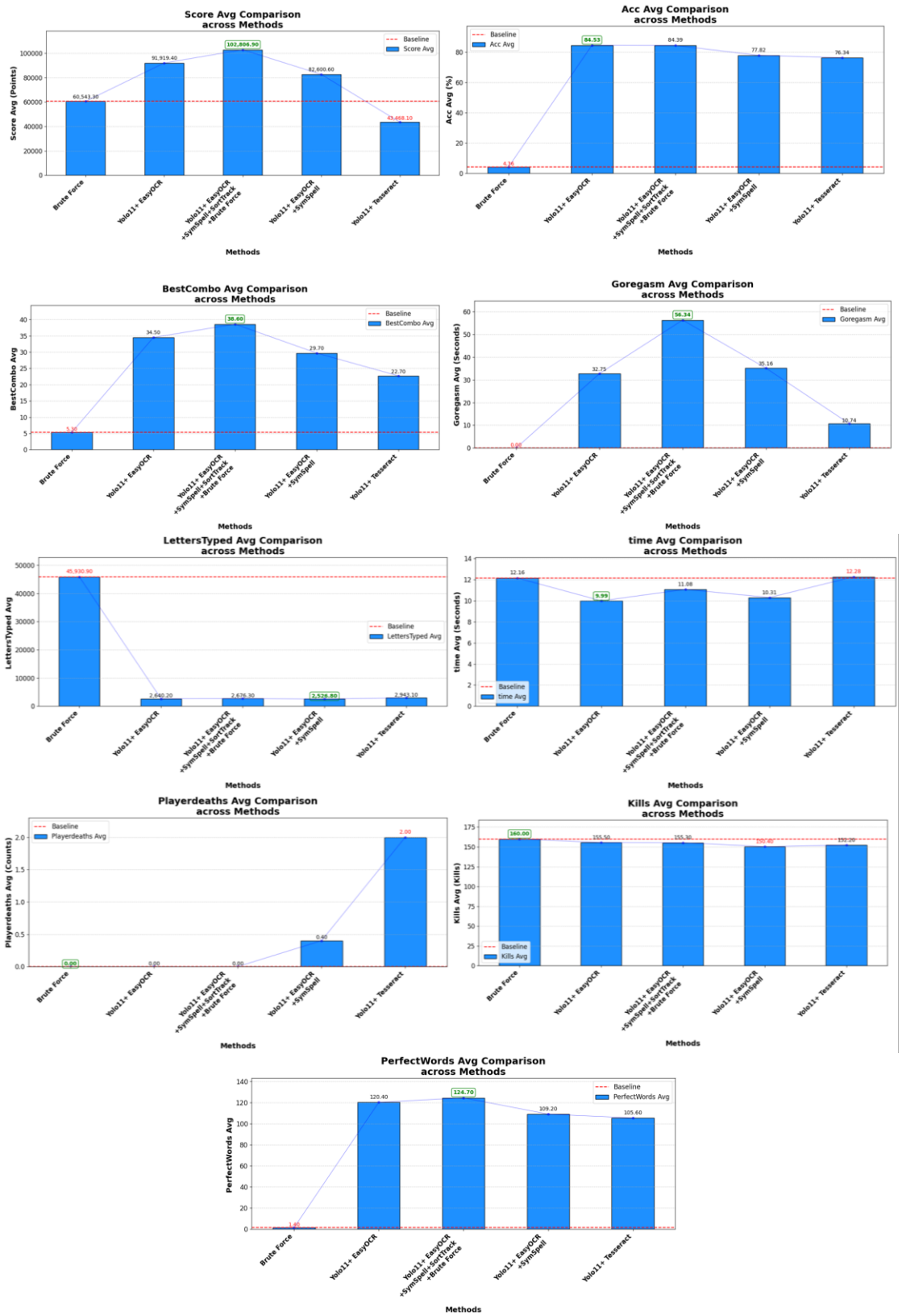


Figure 7 presents a comparative analysis of average gameplay metrics across all methods, visualized for metrics such as Score Avg, Acc Avg, Best Combo Avg, Goregasm Avg, Letters Typed Avg, Time Avg, Player Deaths Avg, Kills Avg, and Perfect Words Avg. Each chart highlights the relative performance of methods against the baseline for key gameplay outcomes.

Table 4 presents the throughput (FPS) of different methods across various game maps, highlighting variations in performance by level.

Method	Level	Throughput (FPS)
Yolo11+ EasyOCR	Map4	19.1
Yolo11+Tesseract	Map4	13.68
Yolo11+EasyOCR+SymSpell	Map4	18.05
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	Map4	16.16
Yolo11+ EasyOCR	Map5	19.27
Yolo11+Tesseract	Map5	16.7
Yolo11+EasyOCR+SymSpell	Map5	18.83
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	Map5	18.38
Yolo11+ EasyOCR	Map6	18.6
Yolo11+Tesseract	Map6	16.97
Yolo11+EasyOCR+SymSpell	Map6	18.18
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	Map6	16.7
Yolo11+ EasyOCR	Map7	19.72
Yolo11+Tesseract	Map7	17.45
Yolo11+EasyOCR+SymSpell	Map7	19.17
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	Map7	15.26
Yolo11+ EasyOCR	Map8	17.2
Yolo11+Tesseract	Map8	16.73
Yolo11+EasyOCR+SymSpell	Map8	16.9
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	Map8	15.22

Table 5 summarizes the average throughput (FPS) for each method, providing an overall comparison of efficiency.

Method	Throughput (FPS)
Yolo11+ EasyOCR	18.778
Yolo11+EasyOCR+SymSpell	18.226
Yolo11+ EasyOCR+SymSpell+SortTrack+Brute Force	16.344
Yolo11+Tesseract	16.306

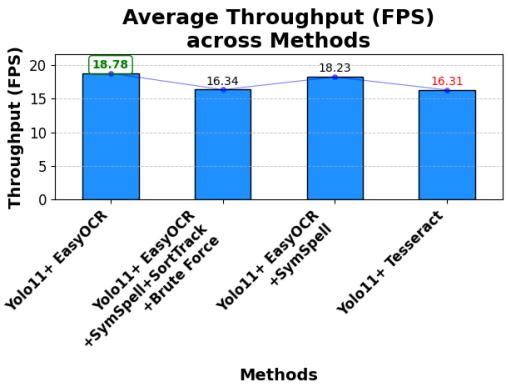


Figure 8 illustrates the average throughput (FPS) for different methods, showcasing their computational efficiency during gameplay automation

The experimental results highlight the comparative performance of five AI-driven methods across multiple gameplay metrics and computational efficiency. The key evaluated factors include Score Average (Score Avg), Accuracy Average (Acc Avg), Letters Typed Average (LettersTyped Avg), Player Deaths Average (Playerdeaths Avg), and Average Time per Level (Time Avg). Additionally, Throughput (FPS) was analyzed to assess computational efficiency across various levels. The results are summarized in Table 3, Table 4, and Table 5, with visual representations provided in Figures 7 and 8.

5.1 Performance Analysis by Metric

Table 3 presents a comparative evaluation of average gameplay metrics across all methods. The YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force method achieved the highest Score Avg (102,806.90), demonstrating its superior performance in gameplay efficiency. YOLOv11 + EasyOCR followed closely with a score of 91,919.40, while the Brute Force method recorded the lowest score (60,543.30) due to its exhaustive search approach.

In terms of Accuracy (Acc Avg), the YOLOv11 + EasyOCR method achieved the highest recognition accuracy (84.53%), slightly outperforming YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force (84.39%). In contrast, the Brute Force approach had the lowest accuracy (4.36%), as expected from its random character guessing mechanism.

The Letters Typed Avg metric, which indicates typing efficiency, is also summarized in Table 3. The YOLOv11 + EasyOCR + SymSpell method had the lowest average keystrokes (2,526.80), demonstrating the effectiveness of SymSpell in reducing redundant inputs. Conversely, Brute Force exhibited the highest keystrokes (45,930.90), reinforcing its inefficiency in optimized text input.

For Player Deaths Avg, YOLOv11 + EasyOCR + SymSpell and YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force exhibited minimal player deaths (0.40 and 0.00, respectively), indicating their stability and reliability in gameplay automation. YOLOv11 + Tesseract, however, had the highest player deaths (2.00), highlighting its lower recognition accuracy under fast-paced conditions.

The Time Avg per level comparison (Table 3) shows that YOLOv11 + EasyOCR performed the fastest, completing levels in 9.99 seconds, while Brute Force took the longest (12.16 seconds) due to its inefficient method of trial-and-error typing.

5.2 Method-Specific Observations

The computational performance of each method was evaluated based on Throughput (FPS) across different game levels, as presented in Table 4 and Table 5. Table 4 details the FPS variations per level, while Table 5 summarizes the average FPS for each method.

The YOLOv11 + EasyOCR method achieved the highest average throughput (18.778 FPS), demonstrating its efficiency in processing frames quickly without additional computational overhead. The YOLOv11 + EasyOCR + SymSpell method followed closely with 18.226 FPS, balancing speed with improved text recognition accuracy through error correction. The YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force method recorded 16.344 FPS, slightly lower due to the added complexity of SORT tracking and brute-force fallback mechanisms. Meanwhile, YOLOv11 + Tesseract had the lowest throughput (16.306 FPS), highlighting its processing limitations in real-time text recognition.

Figure 7 provides a visual comparison of average gameplay metrics across all methods, highlighting differences in score, accuracy, letters typed, time per level, and player deaths. Figure 8 illustrates the average FPS achieved by each method, emphasizing their computational efficiency.

6. Conclusion

This study investigated various AI-driven methodologies for automating in-game text recognition and input within a high-speed, dynamic gaming environment. Five distinct approaches—Brute Force, YOLOv11 + EasyOCR, YOLOv11 + Tesseract, YOLOv11 + EasyOCR + SymSpell, and YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force—were systematically evaluated across multiple performance metrics, including accuracy, efficiency, gameplay success, and computational throughput (FPS). The objective was to identify the most effective approach for real-time text recognition and automated input in fast-paced scenarios.

The experimental results demonstrate that YOLOv11 + EasyOCR + SymSpell + SORT + Brute Force achieved the highest overall score (102,806.90) and excelled in key gameplay metrics such as accuracy (84.39%), streak consistency, and combo-based performance. The integration of SymSpell for text correction, SORT for object tracking, and a brute-force fallback mechanism contributed to its robust performance. However, its added complexity slightly reduced processing speed, resulting in an average throughput of 16.344 FPS—sufficient for real-time gameplay but lower than more streamlined methods.

The YOLOv11 + EasyOCR approach emerged as the most computationally efficient, achieving the highest throughput (18.778 FPS) while maintaining strong accuracy (84.53%) and rapid text recognition. This method demonstrated an optimal balance between speed and accuracy, making it highly suitable for real-time automation tasks where computational overhead must be minimized. The YOLOv11 + EasyOCR + SymSpell method, which integrated error correction, further improved typing efficiency but incurred a slight reduction in throughput (18.226 FPS).

Conversely, the YOLOv11 + Tesseract approach exhibited the lowest performance, with the lowest score (43,468.10), lower accuracy (76.34%), and the lowest FPS (16.306 FPS), highlighting its inefficiency in fast-paced text recognition. The Brute Force method, while functional in eliminating enemies, was the least effective in all other aspects due to its exhaustive search nature.

These findings emphasize the importance of integrating OCR, object detection, real-time tracking, and error correction to achieve effective automation in dynamic environments. Beyond gaming, the proposed AI framework has potential applications in automated license plate recognition (ALPR), AI-driven UI automation, and OCR-based quality assurance (QA) systems. Future research could explore optimizing model architectures and expanding dataset diversity to further enhance adaptability in real-world scenarios.

References

[1] P. N. Thai, T. Suzuki, and K. Aizawa, "PLayerTV: Advanced Player Tracking and Identification for Automatic Soccer Highlight Clips," arXiv preprint arXiv:2407.16076, Jul. 2024.

[2] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, "Play and Learn: Using Video Games to Train Computer Vision Models," arXiv preprint arXiv:1608.01745, Aug. 2016.

[3] M. K. Prajapati, N. H. Bhalodiya, and V. K. Dabhi, "AUTOMATIC LICENSE PLATE RECOGNITION USING YOLOV4 AND TESSERACT OCR," International Journal of Creative Research Thoughts, vol. 10, no. 5, pp. 389-393, May 2022.

[4] M. Kim, "Easy-Yolo-OCR," GitHub repository, 2023. [Online]. Available: https://github.com/aqntks/Easy-Yolo-OCR

[5] M. R. Apriyadi and D. Antoni, "Enhancing Automated Vehicle License Plate Recognition with YOLOv8 and EasyOCR," Indonesian Journal of Information Systems, vol. 6, no. 1, pp. 25-34, 2023.

[6] S. Kumar, R. Kumar, and P. K. Sharma, "Intelligent automation of invoice parsing using computer vision techniques," Multimedia Tools and Applications, vol. 81, no. 28, pp. 40419-40437, Dec. 2022.

[7] R. K. Pandey, S. Sharma, and A. Kumar, "Comparison of Image Preprocessing Techniques for Vehicle License Plate Recognition Using OCR: Performance and Accuracy Evaluation," arXiv preprint arXiv:2410.13622, Oct. 2023.

[8] A. Wang, Y. Zhang, and C. Li, "YOLOv11: An Overview of the Key Architectural Enhancements," arXiv preprint arXiv:2410.17725, Oct. 2023.

[9] S. Ranathunga, P. Liyanage, and D. Dias, "Adapting the Tesseract Open-Source OCR Engine for Tamil and Sinhala Legacy Fonts and Creating a Parallel Corpus for Tamil-Sinhala-English," arXiv preprint arXiv:2109.05952, Sep. 2021.

[10] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple Online and Realtime Tracking," arXiv preprint arXiv:1602.00763, Feb. 2016.

[11] L. Perez and J. Wang, "The Effectiveness of Data Augmentation in Image Classification using Deep Learning," arXiv preprint arXiv:1712.04621, Dec. 2017.

[12] W. Garbe, "SymSpell: A Very Fast Spell Checking and Correction Algorithm," GitHub repository, 2023. [Online]. Available: https://github.com/wolfgarbe/SymSpell

[13] R. Smith, "An Overview of the Tesseract OCR Engine," in Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), Curitiba, Brazil, 2007, pp. 629-633, doi: 10.1109/ICDAR.2007.4376991.

[14] D. R. Vedhaviyassh, R. Sudhan, G. Saranya, M. Safa, and D. Arun, "Comparative Analysis of EasyOCR and TesseractOCR for Automatic License Plate Recognition using Deep Learning Algorithm," in 2022 6th International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2022, pp. 966-971, doi: 10.1109/ICECA55336.2022.10009215.

[15] M. Du, P. He, X. Li, C. Zheng, and X. Zhou, "PP-OCR: A Practical Ultra Lightweight OCR System," arXiv preprint arXiv:2009.09941, Sep. 2020.