Movement Hacks in Video Games using Visual Input Only

Yvette Lin
Stanford University
yvelin@stanford.edu

William Song Liu
Stanford University
wiliu@stanford.edu

Abstract

The popularity of video games worldwide means that inevitably, some players will turn to scripts/hacks to cheat. To combat this, games implement a wide variety of anticheat measures that prevent tampering with game files or reading memory. However, a cheat which relies on visual only input is difficult to detect or prevent, since it only information already available to the player. To this end, real-time object detection models such as YOLO offer a promising approach. However, finetuning these models is challenging, as it usually requires hand-annotating game objects visually unique to each video game. To this end, we propose an approach to building vision-only video game hacks by leveraging a segmentation foundation model for dataset creation and finetuning an object detection model, which can be used for the downstream task of the video game cheat. This approach avoids manually hand-annotating data and requires human input on as little as a single frame of gameplay video. We demonstrate our approach on the task of dodging enemy projectiles in the popular video game League of Legends, successfully creating a cheat that is able to perform correct movement inputs to avoid enemy projectiles based on visual information only. Through quantitative and qualitative evaluations, we demonstrate the feasibility of this approach, hopefully providing directions for the improvement of modern anticheat systems.

1. Introduction

A problem that every online multiplayer video game needs to face is the existance of cheaters. If not managed well, rampant cheating can ruin the experience of legitimate players, leading to poor reviews and overall frustration. To combat this, most video games implement some form of anticheat. These measures can range from encrypting the code base to prevent reverse engineering, file hashing to verify the integrity of files, obfuscating memory to prevent the reading of game values during play, and even detect and block other programs that may be trying to interfere with the game process [12]. Even with all of these measures in place, there is a type of cheat program that may be much harder to detect than others—a program that does not need to read or write from memory or the game files at all, a visual-only cheat. In this paper, we aim to demonstrate the ease and practicality of creating such a cheat on a single-player game, in order to encourage the development of more advanced anticheat methods.

We propose a machine-learning approach to automate a common task found across many top-down video games—dodging projectiles—using only visual information. A depiction of this task is shown in Figure 1; namely, the player character (right) must input a movement command to avoid a dangerous projectile cast by the enemy character (left). Precisely, we propose to fine-tune the real-time object detection foundation model You Only Look Once (YOLO) [17] to detect elements in the game, including the player, enemies, and projectiles.

Figure 1. Example task, projectile to dodge. The enemy character (left) casts a dangerous projectile rightward toward the player character (right). To avoid being hit, the player in this example should move up or down to move out of the projectile's path. This example is taken from the popular video game League of Legends.

Object detection approaches that are similar to ours (using a finetuned foundation model such as YOLO) have been previously demonstrated for aimbots for first-person shooter (FPS) games [4, 5, 10, 18], but our work addresses a distinct challenges compared to previous projects. The objects of interest in FPS games tend to be humanoid, so a

pretrained human figure detector has the potential to work out of the box. Additionally, when these methods require further finetuning, they are finetuned on hand-labeled data, which is a time-consuming process.

To summarize our problem, our input is frames of a video game as would be displayed on screen for a human player, possibly in a game state where a dangerous projectile is being thrown at the player character, and our desired output is the correct sequence of actions (movement input) to avoid such a projectile if necessary, with the constraint that we do not have access to hand-labelled training data a priori, and want to limit the amount of additional human supervision needed to train our model.

To these challenges, we propose to leverage the power of a segmentation foundation model such as the Segment Anything Model (SAM) [11]. We use SAM to automatically segement out the objects of interest and propose a data processing pipeline that generates training data requiring human supervision on object classification on just one frame of a short gameplay video. We use this training data to finetune our object detector, which then provides the visual information needed to perform our end task of dodging projectiles.

We demonstrate our method on the task of dodging "skillshots" (projectiles thrown by enemy players) in the popular MOBA game League of Legends. We evaluate the objection detection and task performance capabilities of our method, demonstrating the possibility of easily creating automated agents to cheat at this common task in video gaming without the need for hand-labeled data. Through our work, we hope to raise awareness about current cheat capabilities and provide directions in which anticheat systems should improve.

2. Related Works

2.1. Deep Learning to Play Video Games

There have been many prior visual video game playing projects, the most notable of which include DeepMind's invention of the Deep-Q Learning (DQN) architecture, which used Convolutional Neural Networks (CNNs) to approximate the optimal policy Q^* in the Atari 2600 games (also known as Atari-57 as there are 57 games) [15]. Since then, numerous improvements have been made to DQNs on the Atari-57 suite of games [8, 2, 1]. While the performance of these models are impressive, Atari games are visually very simple compared to most other games, so it may be difficult to apply the same techniques in modern video games.

A somewhat visually more complex game and series of gameplay agents arose from the Visual Doom AI Competition (ViZDoom), a fast and lightweight implementation of the classic 3D first person shooter game Doom [9]. In the second edition of the competition, automatic object labeling was implemented into the engine, allowing competitors to reference the bounding box, position, rotation, and movement of game objects during training [20]. One such bot, *YanShi*, took advantage of this information to train a Region Proposal Network (RPN) that identified the location of resources and enemies at runtime, obtaining second place in the competition with previously unseen environments [20]. *YanShi* proved the viability of two-module approaches to video game playing tasks, with an object detection module working in tandem with a reinforcement learning module.

2.2. Vision-based Cheats

We find that most of the vision-based cheats with methods available [4, 5, 10, 18] take a similar approach to ours in that they tend to use a finetuned YOLO object detector. However, unlike our work, the majority of the focus using this approach has been on first-person shooter (FPS) games, usually with the goal of creating aimbots. Additionally, our task is potentially more challenging due to the lesser amount of realism in our chosen game: many available pretrained object detectors are capable of detecting humanoid figures, as this is a popular task across multiple domains, compared to the appearance of magical projectiles, which do not have to resemble any type of common object, so are unlikely to be detected out-of-the-box. Additionally, finetuning to a specific game's visuals is still often required, and in the example projects we have knowledge of, this was done on hand-annotated data, potentially making development slow and tedious.

3. Methods

We present a pipeline to train an object detection model to accomplish a novel downstream task in video games (here, dodging enemy projectiles), without the need to tediously hand-annotate data, which would be normally be required to finetune a model to recognize usually out-of-original-distribution game objects. Our insight is that we may leverage a segmentation foundation model such as SAM [11] to perform most of the data annotation that would normally be required, resulting in a dataset creation step that in our approach requires minimal human supervision. We now describe our proposed approach end-to-end, from dataset creation (Section 3.1), to training the model (Section 3.2), to the resulting movement input (Section 3.3) as a result of the object detection. Figure 2 presents an overview of our approach.

3.1. Dataset Creation from Segmentation

To obtain the data using to finetune our YOLO object detector, we start with a short screen-recorded gameplay video gathered by one of the authors. First, we crop the video to remove some of the user interface elements that are irrelevant to our task. We use SAM to automatically generate

(a) Reference segment (b) Segment train image (c) Annotate train image (d) Train object detector

Figure 2. Dataset creation and finetuning object detection. (a) First, we segment each frame and choose a single (or small number of) frame(s) where the projectile is visible. We use the corresponding object mask to compute heuristics $\mathcal{H}$. (b) Now, to prepare the dataset, we segment each of the train/val/test images and (c) computationally identify and annotate the objects of interest using similarity in the heuristics $\mathcal{H}$. (d) We now finetune the YOLO object detector to detect projectiles, informing the player of the correct movement input to dodge.

segments $\left\{M_t^{(1)}, \ldots, M_t^{(K_t)}\right\}$ for each frame $\tilde{X}_t$. We arbitrarily choose a single (or small number of) frame(s) $\tilde{X}^*$ where a projectile is visible and manually identify the segment M^* corresponding to the projectile. Hence, for this single frame $\tilde{X}^*$, we now obtain an annotation for our object of interest, allowing it to be used as training data to finetune the YOLO object detector. This step is depicted in Figure 2(a).

We want to now extend this to the remaining frames of the video: concretely, for each $\tilde{X}_t \in \{\tilde{X}_1, \ldots, \tilde{X}_{\tilde{T}}\} \setminus \tilde{X}^*$, if it exists in that frame $\tilde{X}_t$, we wish to find $M_t^* \in \left\{M_t^{(1)}, \ldots, M_t^{(K_t)}\right\}$ the segment that corresponds to the same type of object (a projectile) as M^*. (Note that this does not need to necessarily be the same game object as M^*, as many such projectiles may be fired over the course the video; instead, M^* serves as a visual reference to identify M_t^*.) To do this, we identify the segment $M_t^* \in \left\{M_t^{(1)}, \ldots, M_t^{(K_t)}\right\}$ that most resembles M^* according to a set of heuristics $\mathcal{H}$, given that it is sufficiently close to M^* for each $h_j \in \mathcal{H}$; else, we assume that a projectile is not present in $\tilde{X}_t$. Concretely,

$$M_t^* = \underset{M_t^{(i)} \in \left\{M_t^{(1)}, \ldots, M_t^{(K_t)}\right\}}{\operatorname{argmin}} \sum_{h_j \in \mathcal{H}} d_j(h_j(M^*), h_j(M_t^{(i)})) \tag{1}$$

where $d_j(\cdot, \cdot)$ is a distance metric (we choose the L1 distance for all j), given that

$$d_j(h_j(M^*), h_j(M_t^*)) < \delta_j \tag{2}$$

for all j for a minimum threshold δ_j. Otherwise, $M_t^* = \varnothing$. We found the set of heuristics $\mathcal{H} = \{\textsc{Area}, \textsc{AverageColor}\}$ to work well. These steps are depicted in Figure 2(b,c).

Additionally, to avoid false positives when using this heuristic approach, we prefer to set the thresholds δ_j lower rather than higher. We then additionally perform a *discard step* where we discard all the frames where a segment corresponding to the projectile was not found, thus removing all false negatives, and keep the remaining frames for use in our dataset. This step has the potential to introduce greater mismatch between the true distribution of data and our dataset, but empirically we find that this increases the performance of the object detector—i.e. incorrect annotations are more harmful than lack of data.

Finally, after applying this data processing, we obtain an annotated training data set that is used to finetune the YOLO object detector, with human supervision only explicitly needed on one image.

3.2. Finetuning Object Detector

We finetune the pretrained YOLOv8n model [6] with the training data from Section 3.1 to obtain an object detector capable of recognizing projectiles in gameplay images. This step is depicted in Figure 2(d). YOLOv8n is pretrained on the COCO dataset [14] and is chosen over the other available pretrained models from [6] for fastest inference time. Additionally before training, we apply to our created dataset (the output of Section 3.1) various data augmentation methods (random crop, scale, flip, color jitter) provided in the training pipeline of [6].

3.3. Player Movement

Now, at inference time, consider a short video sequence of recent frames output from the video game $\{X_1, \ldots, X_T\}$ in which a projectile is consecutively visible for all frames. For each frame $X_t \in \{X_1, \ldots, X_T\}$, we obtain the location p_t^{proj} of the projectile via the finetuned object detector. We assume that the player character is at a fixed known loca-

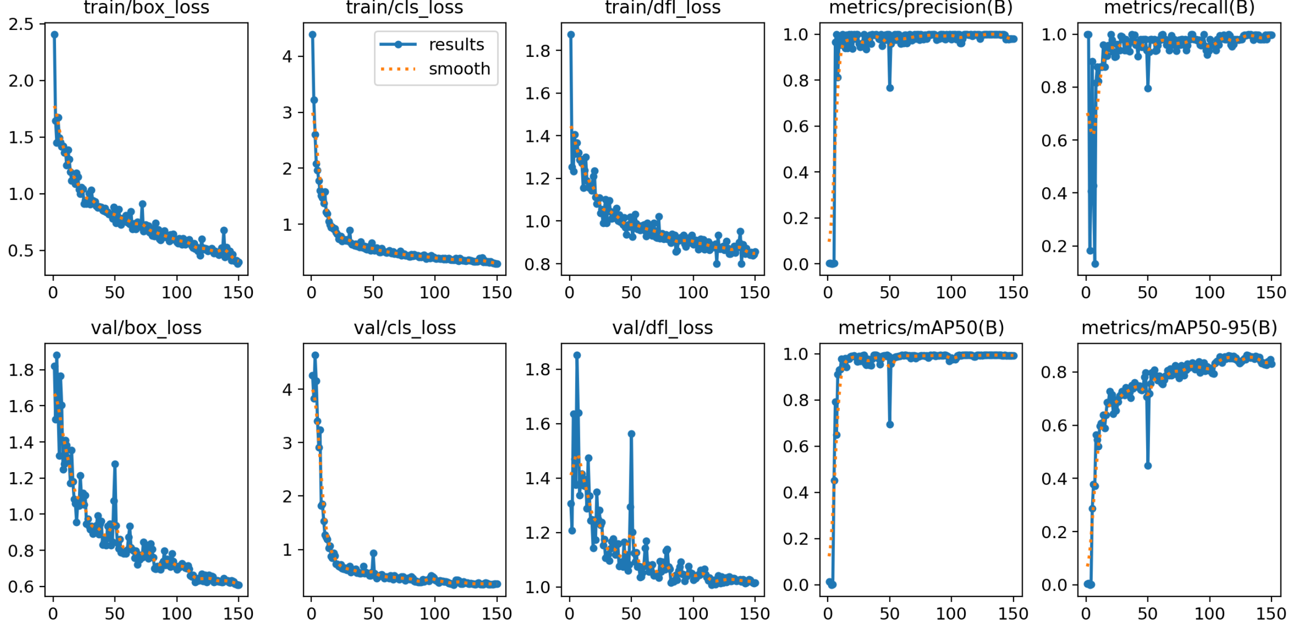

Figure 3. Metrics vs. number of train epochs. We show the bounding box loss, classification loss, and Distribution Focal Loss (DFL) for the train and validation sets, as well as the precision, recall, and mean average precision at a threshold of 0.5 (mAP50) and across the range of 0.5 to 0.95 (mAP50-95).

tion p^{char}throughout all frames (this is easily be satisfied in our case by placing the game in so-called "locked camera" mode).

The velocity of the projectile v_T^{proj} relative to our character is estimated by

$$v_T^{\text{proj}} = \frac{1}{T-1}\sum_{t=1}^{T-1}(p_{t+1}^{\text{proj}} - p_t^{\text{proj}}). \tag{3}$$

Let $\hat{v}_T^{\text{proj}} = v_T^{\text{proj}}/\|v_T^{\text{proj}}\|$. Then

$$t = (p^{\text{char}} - p_T^{\text{proj}}) \cdot \hat{v}_T^{\text{proj}} \tag{4}$$

is the projected length of p^{char} onto the line $p_T^{\text{proj}} + t\hat{v}_T^{\text{proj}}$. If $t < 0$, this indicates that the projectile is travelling away from our character, so we are not in danger of collision.

However, in the case that $t \geq 0$, we are still in danger of collision. In this case, we compute the distance of p^{char} to the line as

$$d = \|(p^{\text{char}} - p_T^{\text{proj}}) - ((p^{\text{char}} - p_T^{\text{proj}}) \cdot \hat{v}_T^{\text{proj}})\hat{v}_T^{\text{proj}}\| \tag{5}$$

If the distance is within some threshold $d < \delta$, where δ is determined by the size of the projectile and character models, the projectile is on course to hit the character, and we must move the character to avoid collision. We choose to simply move the character perpendicular to the direction $\hat{v}_T^{\text{proj}} = (x_T^{\text{proj}}, y_T^{\text{proj}})$ of the projectile, in the direction $(-y_T^{\text{proj}}, x_T^{\text{proj}})$ (note that $(y_T^{\text{proj}}, -x_T^{\text{proj}})$ would also suffice).

4. Experimental Results

4.1. Implementation

To create our dataset according to the method described in Section 3.1, we use ∼2000 frames of screen-recorded gameplay footage created by one of the authors, with an 80%/10%/10% train/val/test split. To segment the images, we used the default SAM hyperparameters, except we reduce the points-per-side from 32 to 16 to prevent the automatic mask generator from creating too finely-grained segments. After discarding, we are left with a final dataset size of around ∼300 images to finetune YOLO.

To finetune the YOLO object detector, we use the default hyperparameters in the [6] implementation and train for 150 additional epochs. We train the object detector on an NVIDIA GeForce RTX 3090 GPU (∼10 minutes); however, once we finish training, we export the model to ONNX [3] to perform inference on CPU.

4.2. Object Detection

We evaluate the performance of our finetuned object detector. Since we don't have ground-truth labels for locations and bounding boxes in our self-collected dataset without labeling these by hand across a large number of frames, which is somewhat tedious, we approach this evaluation question in two different ways.

First, we evaluate the object detection capability of our

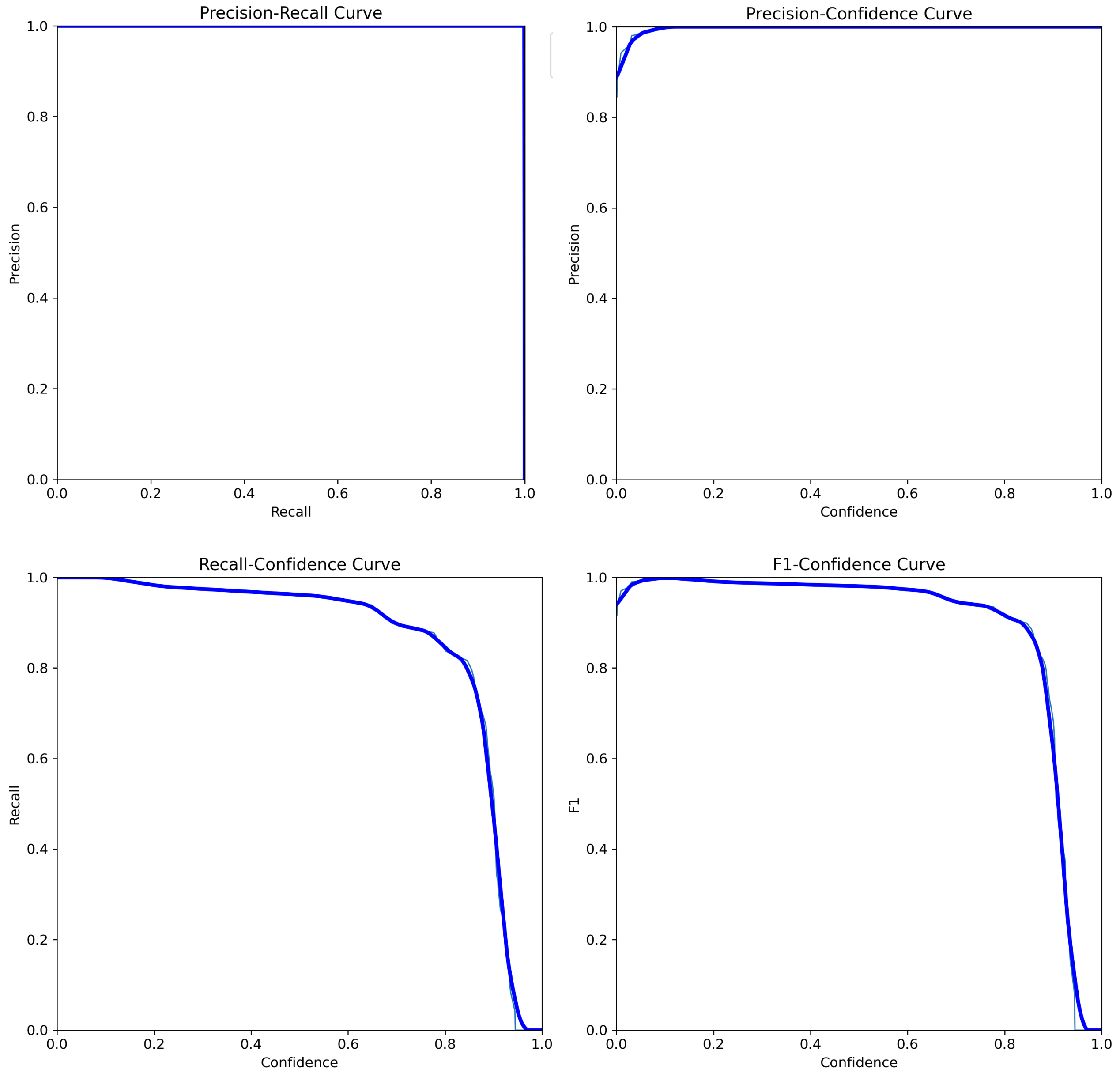

Figure 4. Curves for several metrics. Precision-recall (top left), precision-confidence (top right), recall-confidence (bottom left), and F1-confidence (bottom right) curves.

finetuned detector assuming that the data that it sees (as a result of Section 3.1) is the ground truth. Even if this assumption is not necessarily true in all cases, these evaluations show that our process of finetuning the YOLO object detector indeed results in a model that is able to successfully learn the relationships represented in the data. A reasonable performance here is also an indicator that our dataset creation process results in semantically meaningful labels, rather than pure noise, as it indicates that there exists some signal that the object detector is able to learn.

To this end, in Figure 3 we show the bounding box loss, classification loss, and Distribution Focal Loss (DFL) [13] (which is a loss associated with the distribution of bounding box offsets, capturing uncertainty in box location) for the train and validation sets over the training epochs. Additionally, for these epochs, we show the precision, recall, and two mean average precision (mAP)-based metrics. The

mAP is defined as

$$\text{mAP} = \frac{1}{n} \sum_{k=1}^{n} \text{AP}_k \tag{6}$$

where n is the number of classes and AP_k is the average precision of class k. The mAP50 is the mAP calculated at an intersection over union (IoU) threshold of 0.5. The map50-95 is the average mAP calculated at IoU thresholds across the 0.5 to 0.95 range. We chose to stop the training at 150 epochs to prevent overfitting as this is where the validation losses and metrics begin to plateau.

In Figure 4, we show the precision-recall, precision-confidence, recall-confidence, and F1-confidence curves. In Table 1 we report our final mAP50 and mAP50-95 scores. To our knowledge, there are no existing public or open source baselines on the same task we are trying to accomplish, so just to provide a point of comparison, we make a comparison to YOLOv8n and YOLOv8x (the most powerful pretrained model provided by [6]), on the COCO dataset. Again note this is not meant to be a fair comparison, since the COCO task is much more difficult with 80 different classes, but just meant to provide a sense of scale for this metric.

Method	mAP50 (↑)	mAP50-95 (↑)
Ours, on created dataset	99.5	86.4
YOLOv8n, on COCO	–	37.3
YOLOv8x, on COCO	–	53.9

Table 1. Mean average precision (mAP50 and mAP50-95) for our model. Due to the lack of existing baselines, we provide metrics of the base model (YOLOv8n) and a more powerful version (YOLOv8x) on their original task (COCO). Note that this is not meant to be a fair comparison, but to just provide a sense of scale.

Additionally, we show the normalized confusion matrix in Figure 5. All together, these metrics show strong performance on the created dataset, validating that our finetuned model is able to learn the relationships in the data, and that there is a strong underlying signal present in our data.

Second, we report the *binary accuracy* of the detector in terms of the proportion of frames in which the detector is simply able to correctly report whether there exists a projectile in the frame, as it is much easier to engineer/label a test set ground truth for this by hand. In practice, we do this by picking equal-length video clips where a projectile is visible and not visible for the entire duration, with an equal number of frames where a projectile is/isn't visible.

In Table 2 we show results for the object detection capability of our finetuned object detector using this *binary accuracy* metric. We compare against the baseline of a YOLO object detector with no finetuning. Since the pretrained YOLO detector without finetuning has no understanding of

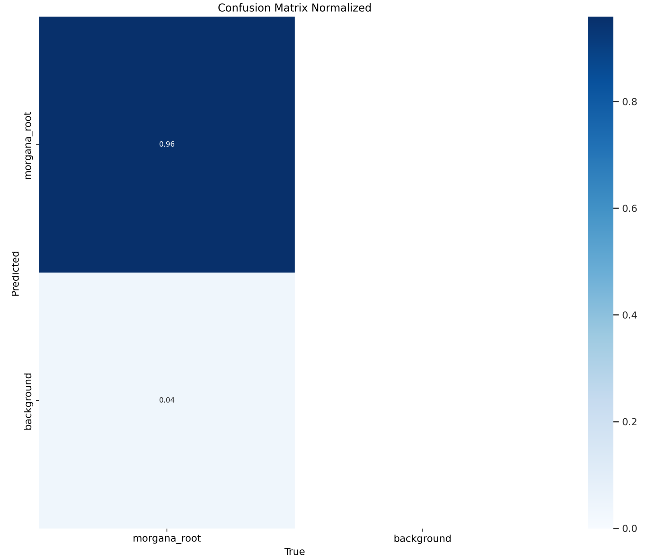

Figure 5. Normalized confusion matrix.

what the projectile is supposed to look like since it is out of distribution of the original dataset using for pretraining, it is unable to detect the projectile at all, so this result is expected. Our finetuned detector shows stronger performance.

Method	Binary Accuracy (↑)
Ours (w/ finetuning)	95.8%
YOLOv8n (w/o finetuning)	0.0%

Table 2. Binary accuracy of object detection. This is the proportion of frames where the binary presence (is/is not present in the frame) of the projectile was detected correctly.

Qualitatively, we note that the failure cases tend to be false negatives (projectile is in frame but not detected) where the projectile is too close to the edge of the screen or to the enemy character. Examples of these are shown in Figure 6. In practice, we do not expect this to be a problem for our movement strategy, since the projectile just needs to be visible for a sufficient number of frames to trigger the player character, which is located near the center of the screen, to move.

We additionally perform an ablation test to validate the *discard step* described in Section 3.1. Table 3 shows the mAP50 and mAP50-95 for our object detector with and without this step, showing that this step indeed improves performance.

4.3. Movement

We wish to evaluate whether given the detected objects, our method is able to successfully evade thrown projectiles. Note that, to our knowledge, there do not exist any publicly available open-source cheats or scripts to accomplish our

Figure 6. Example success and failure cases. Left: a successful detection. Middle: an example failure, caused by the projectile being too close to the character that fired it. Right: an example failure, caused by the projectile being clipped by the edge of the screen.

Method	mAP50 (↑)	mAP50-95 (↑)
Ours	99.5	86.4
Ours w/o discard step	85.9	74.1

Table 3. Mean average precision (mAP50 and mAP50-95) w/ and w/o *discard step*.

Trial distance	Proportion dodged
Long-range	90%
Short-range	0%

Table 4. Proportion of projectiles dodged. The trials fall into two categories, *long-range* and *short-range*, describing the distance between the human player and the cheating player

specific end task of dodging League of Legends skillshots, so it is difficult to compare performance on our end task to a previously published baseline.

To evaluate our method, we have a character controlled by our movement cheat play vs. a human player. The two players move around and vary their position around the map across this experiment. For each trial the human player then fires a projectile directly at the cheating player. We report the proportion of times the cheating player is able to successfully avoid being struck by the projectile in Table 4. Note that we split the trials into two categories, *long-range* and *short-range*. *Long-range* refers to trials in which the two players were at a distance apart that is at or near the maximum travel distance of the projectile ability, and *short-range* refers to all other cases. We report our results this way because we notice qualitatively that while our method successfully detects the projectile most of the time in all cases (both long-range and *short-range*), the cheat is a bit too slow to move the character fully out of the way before the projectile hits if the projectile travel distance is lower. Addressing this limitation by improving the speed of either model inference or game interfacing is a possible direction for future work.

Additionally, we provide a video supplement showing examples of our cheat at work.

Undetectability? Certainly not definitive proof, but the authors of this paper did not receive any bans or suspensions on their League of Legends accounts throughout working on this paper, despite the use of a kernel-level anticheat in League of Legends, which perhaps points in the favor of the undetectability of this method by current anticheat systems.

5. Conclusion

In this work, we propose an approach to creating difficult-to-detect cheats for video games based on visual information only, while minimizing the tedious process of hand-annotating data to for objects unique to each video game to finetune an object detection model. Our insight is that a segmentation foundation model can be used to greatly speed up and automate the annotation process. We demonstrate the feasibility of this approach on the task of automatically dodging projectiles in the popular video game League of Legends. While there have been vision-based cheats for FPS games, to our knowledge, we are the first (at least in an academic setting) to tackle this task with this approach.

5.1. Limitations and Future Work

While we demonstrate our method on a specific video game, our method should be extensible to a variety of tasks across different games, which might be an exciting avenue for future work. However, one limitation of our method is

that we are able to use fairly simple heuristics in the dataset creation step due to the relative visual simplicity of League of Legends being a top-down game. Extending this approach to very visually complex games will require more complex heuristics; one possibility is using CLIP [16] similarity. Another limitation is that our input movement commands through mouse movements and actions remain quite "inhuman" and thus could flag our account for suspicion by a sophisticated enough anticheat system. One interesting direction for future work would be to combine with machine learning-based methods aimed at generating human-like behavior, such as in [7] and [19].

Contributions and Acknowledgements

Yvette designed and implemented the dataset creation and YOLO finetuning pipeline. William implemented the movement strategy and interfacing inputs/outputs to the game.

References

[1] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell. Agent57: Outperforming the atari human benchmark. In *International conference on machine learning*, pages 507–517. PMLR, 2020.

[2] A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, et al. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

[3] J. Bai, F. Lu, K. Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[4] Y. Cheng. Character detection in first person shooter game scenes using yolo-v5 and yolo-v7 networks. In *2023 2nd International Conference on Data Analytics, Computing and Artificial Intelligence (ICDACAI)*, pages 825–831. IEEE, 2023.

[5] Y. Cui, M. Si, and Q. Li. Game image detection and application based on improved yolov5. In *2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*, pages 1012–1017. IEEE, 2023.

[6] G. Jocher, A. Chaurasia, and J. Qiu. Ultralytics YOLO, Jan. 2023.

[7] A. Kanervisto, T. Kinnunen, and V. Hautamaki. Gan-aimbots: Using machine learning for cheating in first person shooters. *IEEE Transactions on Games*, PP:1–1, 01 2022.

[8] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[9] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *2016 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE, 2016.

[10] O. Kermad. Neuralbot. `https://github.com/kermado/NeuralBot`, 2020.

[11] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick. Segment anything, 2023.

[12] S. Lehtonen et al. Comparative study of anti-cheat methods in video games. *University of Helsinki, Faculty of Science*, 2020.

[13] X. Li, W. Wang, L. Wu, S. Chen, X. Hu, J. Li, J. Tang, and J. Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection, 2020.

[14] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, 2014.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[16] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision, 2021.

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2016.

[18] RootKit-Org. Ai-aimbot. `https://github.com/RootKit-Org/AI-Aimbot`, 2024.

[19] T. Witschel and C. Wressnegger. Aim low, shoot high: evading aimbot detectors by mimicking user behavior. In *Proceedings of the 13th European workshop on Systems Security*, pages 19–24, 2020.

[20] M. Wydmuch, M. Kempka, and W. Jaśkowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 11(3):248–259, 2018.