EFFICIENTLY IMITATING HUMAN MOVEMENT IN COUNTER-STRIKE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David Benjamin Durst
August 2024

This dissertation is online at: https://purl.stanford.edu/yz173qh1790

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kayvon Fatahalian, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Human-like agents have the potential to drastically improve multiplayer, first-person shooter (FPS) games. They can serve as engaging teammates, useful practice partners, and anti-social behavior detectors. However, it is difficult to create a multiplayer FPS agent that replicates human behavior, and in particular human movement. Despite game developers' efforts for decades, agents either struggle with the wide range of possible game situations in a multiplayer FPS, or are too computationally inefficient to deploy in a commercial title.

This dissertation contributes a machine learning (ML)-based multiplayer FPS agent that has some of the most human-like behavior demonstrated to date while also satisfying games' performance constraints. Previous game developers avoided ML agent techniques due to the computational requirements. In order to make human-like ML tractable for real-time use in a commercial game, we made three key design decisions. First, we use an imitation learning approach to train the agent. Imitation is the most direct approach for creating human-like agents, and multiplayer FPS titles generate large datasets of demonstrations to imitate. Second, long-term human-like behavior emerges from our ML model's short-term predictions. Predicting only the next few actions enables us to automatically generate large collections of labels and utilize a simple, supervised training process. Finally, we only utilize learning where necessary. Our agent's hybrid architecture utilizes rule-based behavior generators where possible, and the ML model's game state input is in a symbolic format that can be efficiently processed.

We utilize these principles to create a complete agent system. We will describe the system's four components. First, a dataset curation pipeline for creating a large-scale dataset of human movement. Second, an efficient, transformer-based movement model trained to imitate the dataset. Third, a complete agent, known as MLMove, that utilizes the learned movement model to play the multiplayer FPS game Counter-Strike. Finally, we evaluate if MLMove is human-like. Since "humanness" is difficult to evaluate, we present a suite of evaluations, including a user study and large-scale analytics, demonstrating that MLMove's behavior is more human-like than strong baselines including industry-standard and expert-crafted agents. We discuss our agent's limitations and the impact of our work on the wider field of video game agents.

Acknowledgments

First, I would like to thank Kayvon Fatahalian and Pat Hanrahan. Your guidance allowed me to explore a wide range of topics and identify the area where I would have the most impact. You helped me learn the difference between exciting technology and fundamental advances that will change Computer Science. It is amazing that our arguments about ostensibly simple topics, like the relationship between Counter-Strike cover and how crows protect their children from hawks, evolved into a dissertation on efficient machine learning for creating human-like agents. As I move on to other topics, I look forward to continuing to utilize the principles that I learned from you.

Thank you to my mentors and collaborators: Gilbert Bernstein, Feng Xie, Marco Patrignani, and Carly Taylor. Aetherling and MLMove would not have been possible without your guidance. You taught me how to structure a research project and clearly communicate the results. My PhD had a lot of twists and turns, and your advice was crucial for staying (approximately) on track.

During my time at Stanford, I had the pleasure of working with additional, wonderful collaborators: Sanjiban Choudhury, Iuri Frosio, Chen Tessler, Joohwan Kim, Matthew Feldman, David Akeley, Andrew Adams, and Shoaib Kamil. Their ideas were invaluable to my research. I also had a lot of fun sharing ideas and hanging out with my labmates: James Hong, Brennan Shacklett, Purvi Goel, Vishnu Sarukkai, Dan Fu, David Yao, Fait Poms, Ravi Teja Mullapudi, Will Crichton, Dillon Huff, Haotian Zhang, Zander Majercik, Ross Daly, Lenny Truong, Caleb Donovick, Rajsekhar Setaluri, Michael Mara, James Thomas, Teguh Hofstee, and James Hegarty. The wider Agile Hardware (AHA) community was a wonderful source of support and interesting ideas, including Gedeon Nyengele, Kavya Sreedhar, Maxwell Strange, Keyi Zhang, Qiaoyi Liu, Zachary Myers, Jeff Setter, Alex Carsello, Christopher Torng, Fredrik Kjolstad, Mark Horowitz, Nestan Tsiskaridze, Rick Bahr, and Stephen Richardson.

Thank you to the Counter-Strike and video game AI communities. Your continuous encouragement inspired me to keep working and realize my work's potential impact.

Finally, thank you to my parents, Cindy and Michael Durst, for your guidance and support before, during, and after my PhD.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Competitive, multiplayer first-person shooters (FPS) are extraordinarily popular. Multiple titles have tens of millions of users every month [20, 111]. In the games, teams of players navigate a 3D world known as a map. Players coordinate their movement with teammates in order to achieve goals like attacking and defending key map regions. They shoot at enemies in order to compete for control of these key regions.

Computer-controlled players, known as agents, have the potential to benefit the multiplayer FPS genre if they can behave in a human-like fashion. Friends are not always online, so human-like agents could ensure that buddies who imitate your friends are always available to play. Human-like agents could be customized to provide a skill-level appropriate experience for training. Once we have a model of normal human behavior, we can use it to define and detect anti-social behavior like cheating [28, 58]. However, all of these possible applications rely on the existence of human-like agents.

Skilled human behavior in multiplayer FPS is simple to describe. Players attack enemies from multiple directions (known as flanking) in order to overwhelm them, help overwhelmed teammates by moving to defend against flanks, and position near walls known as cover to block enemies' bullets. But, these behaviors are complicated to learn. Humans need hundreds or thousands of hours of experience before they can perfectly coordinate flanks with teammates, know when to provide help, and predict which piece of cover best protects against all likely enemy locations.

Similarly, it is hard to create human-like agents that replicate these behaviors. Game developers in other genres have utilized simpler, inhuman agents based on hand-crafted rules for decades [55, 103]. These rules struggle to replicate the complexity of human behavior in multiplayer FPS. Researchers have created multiplayer FPS agents that learn to win in inhuman fashions [57]. But, the goal of a human-like agent is not to win every time. The goal is to imitate the behavior of a skilled human player. These agents struggle to generate human-like behavior, and thus have not achieved significant adoption in multiplayer FPS games, for three reasons:

Complexity of Human Movement. Hand-crafted, rule-based agents remain the prevalent practice in other game genres. However, multiplayer FPS games can generate a vast array of situations due to multiple teams navigating a complex 3D world. Developers have tried to write down behaviors for every possible situation using decision trees. Subsets of tree nodes generate behavior for specific situations. Figure 1.1a is a simple decision tree that took months to create. Real trees might take orders of magnitude more effort. Figure 1.1a does not even show the full complexity of its simple logic. Figure 1.1b shows the nest of logic inside the tree for determining the current situation. Before trees can generate situation-specific behavior, they first need to select the right part of the tree for the current situation. It is extraordinarily difficult to manually write rules for how to (1) identify every situation and then (2) generate the right behavior for the current situation. As a result, hand-crafted agents fail to react appropriately to a diverse set of game situations.

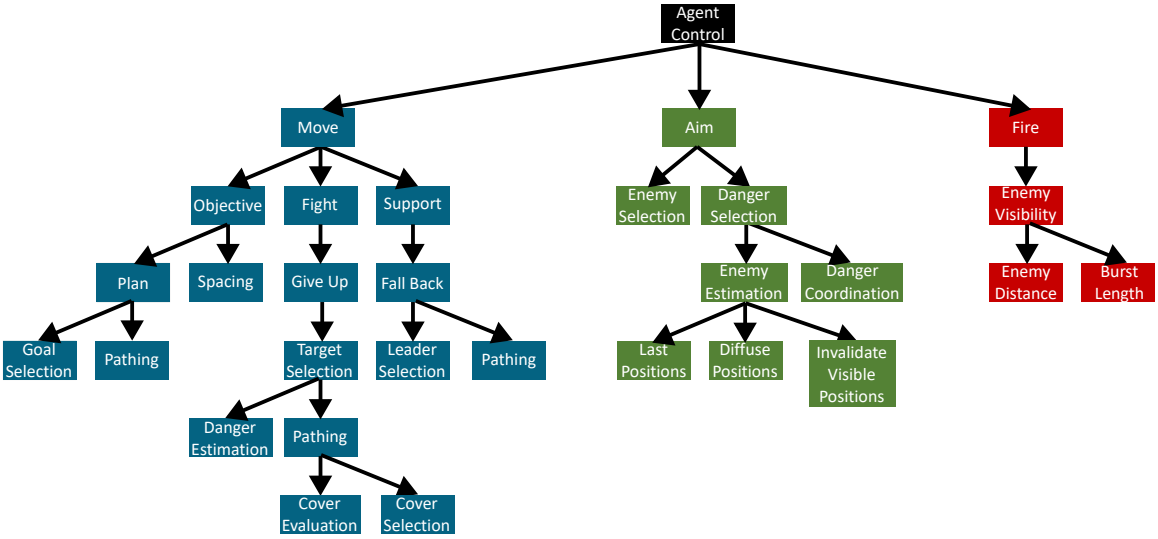
Skill and Human-Like Style are Different. Researchers previously trained superhuman agents that optimize for winning but do not produce human-like movement [4, 57, 62]. Humans expect other players to demonstrate a particular human-like style including collaborative movement techniques and standard map positions. Agents are not exposed to a human-like style of movement and positioning when they learn to win by playing against themselves, and so may learn to win with an inhuman style. Humans frequently do not want to play with skilled agents if the agents' style does not match their expectations [32]. Human-like agents must move in a manner that matches human expectations for their target skill level.

Compute Efficiency. Researchers also previously trained human-like agents from human demonstrations, but these approaches do not satisfy multiplayer FPS games' performance constraints [32, 79]. The key limitations are model size and input game state format. Some of the human-like agents could use large models because they play turn-based games with run-time performance requirements that are orders of magnitude (100×) lower than that of a real-time FPS game. Other human-like agents play multiplayer FPS games, but use rendered game images as input. This approach requires a GPU for every agent to render and process the images, orders of magnitude (800×) more compute than current FPS agents. Most commercial FPS games require agent logic to use only a small fraction of the total per-frame CPU budget [22] (limiting execution to a few milliseconds on a single CPU core [66, 86]). We are not aware of a prior, human-like agent that satisfies the FPS agent performance constraint.

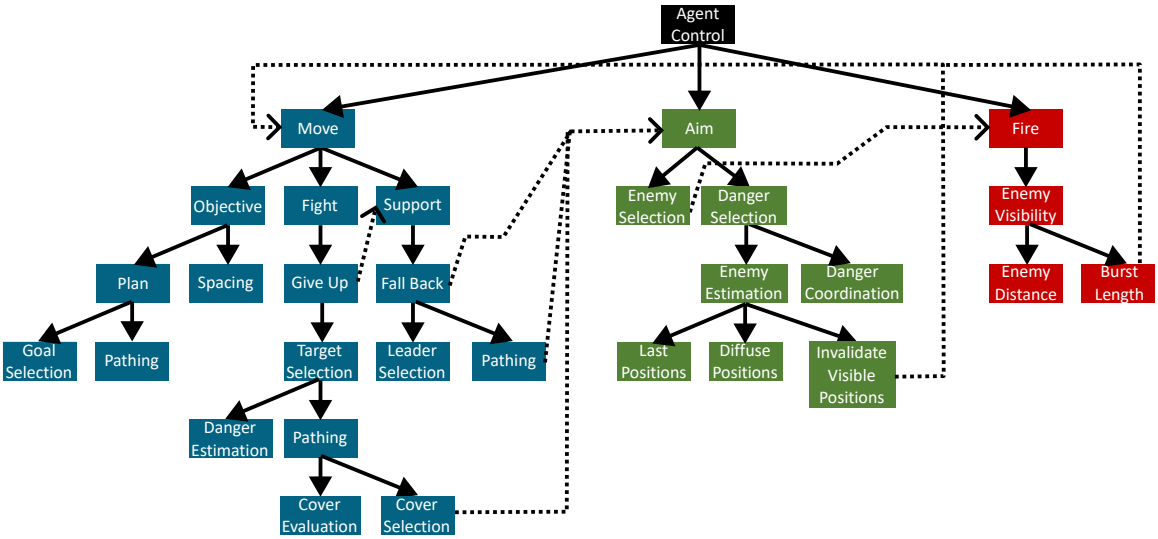
1.1 Key Ideas

The overall contribution of this dissertation is a Counter-Strike agent that (1) has some of the most human-like behavior demonstrated to date (2) while also satisfying FPS games' performance constraints. Three key ideas enable our agent to satisfy the quality and performance goals:

Imitate Human Demonstrations at Scale. If you want to make an agent that demonstrates



(a) This agent uses a tree of hand-crafted rules in order to generate human-like behavior. It uses three sub-trees to address the three different types of behaviors in a multiplayer FPS: moving, aiming, and firing.



(b) The tree includes many hand-authored decisions for when to transition between sub-trees. The dotted lines highlight these transitions.

Figure 1.1: The tree diagrams visualize a hand-crafted agent with and without highlighted state transition logic.

skilled human-like behaviors, the most direct approach is to imitate humans. The main challenge is collecting a dataset of human demonstrations for all possible game situations. Fortunately, multiplayer FPS games are very popular, so they generate large datasets of human demonstration that cover all possible situations. Imitation of these massive logs enables human-like agents that are trained for every possible game situation.

Short-Term Predictions Lead to Long-Term Behavior. Similar to GPT and other LLMs, we found that models which autoregressively predict short-term actions can generate long-term, human-like behavior [85]. This finding has two key benefits. First, it enables us to automatically generate a large collection of reliably correct labels. At each time step in our human demonstration dataset, the labels are just humans' next actions. These labels are more reliable than manually labeled, long-term labels inferring a human's intent over an entire round. Second, we can use a simple model and training procedure. Predicting the next few actions is a standard supervised learning problem. We found that a small model is able to excel at this supervised learning task on our large dataset.

Learning Everything Is Not Necessary. Multiplayer FPS agents must simultaneously aim their weapon, fire bullets, and move around the map. We take a hybrid approach that combines rule-based and learned components, leveraging domain knowledge to ensure each component is efficient. Rule-based components can generate human-like behavior for some aspects of an agent, like aiming and firing. Aiming and firing depend on a subset of game state, so an expert can hand-craft rules that account for the smaller range of possible situations. These rules require very few compute resources. On the other hand, moving is a complex behavior that hand-crafted agents struggle to imitate. As explained above, we take an imitation approach where the model predicts human-like actions from game state. Picking the right game state representation is crucial for model performance. We extract the game state in a compact, symbolic representation. This format enables a small model to efficiently process all players' current states and accurately predict human-like next actions.

1.2 Dissertation Contributions

We use these key ideas to create a compute efficient, data-driven system for creating agents that move like human players in the multiplayer FPS game Counter-Strike. These agents, which include a small transformer-based model [73] trained using imitation learning, move like experienced human team players, execute well-within the agent compute budget of commercial FPS games, and are simple and fast to train. In order to enable the agent creation system, this dissertation makes the following contributions:

(1) Efficient transformer-based movement controller. First, this dissertation presents the first compute-efficient, transformer-based model specialized for controlling movement in Counter-Strike, called MLMove. The model focuses on playing one map (de_dust2) and one game mode

(Retakes). Once trained through standard supervised learning, MLMove produces human-like movement actions in response to evolving game dynamics. Our movement model's amortized runtime cost for controlling two teams of agents in a Counter-Strike match is just under 0.5 ms per game step on a single CPU core (8 ms inference every 16 game steps), meeting commercial game servers' performance requirement. Human evaluators assess that our model's movement is more human-like than both commercially-available agents and expert-crafted rule-based agents by 16% to 59% (according to a TrueSkill rating) in a user study.

(2) Pro-player Counter-Strike movement dataset curation system. Second, this dissertation presents a system for the curation of a 123-hour dataset of Counter-Strike gameplay called CSKnow. This is the first large scale dataset curated for learning team-based movement in a popular FPS game featuring professional players. This curation system demonstrates how to extract game state into a series of tokens that can be used for supervised training of the transformer-based movement model.

(3) Hierarchical agent structure for behavior generation and testing. Third, this dissertation demonstrates how to design a hybrid rule-based/learned agent. We identify that behavior trees, an industry standard technique for designing rule-based agents, are a sufficiently expressive framework for structuring the agent's non-learned behavior generators and integrating the learned movement model. Also, we explain the derived game state features necessary for the agent's rule-based components.

(4) Quantitative positioning metrics for assessing human-like behavior. Fourth, the goal of this dissertation is to produce realistic agent movement at both short-term and longer-term (full round) time scales. The document defines novel quantitative metrics computed on rounds of agent vs. agent self-play that assess how well an agent's movement emulates human players' team-based positioning. We demonstrate these metrics correlate with the human evaluators' assessment of human-like game play.

(5) Interface for controlling agents in a commercial FPS game. Fifth, this dissertation explains the design decisions necessary to train, test, and deploy agents in a commercial game like Counter-Strike. We explain our agent API for reading from and writing to game state. This API enables our efficient agent to run in a CPU-only environment. Our agent API may serve as a guide for future game engines as they improve support for agents.

We demonstrate these contributions in Counter-Strike because it is an excellent platform for human-like agent research. First, Counter-Strike has a large community with a well-defined set of human-like movement strategies. Counter-Strike is one of the seminal titles in the multiplayer FPS genre, with millions of players each day [12]. These expert humans have developed movement strategies. Since we know these strategies exist in our demonstration dataset, we can evaluate if our agent successfully emulates them. Additionally, we can leverage members of the community to take part in user studies. Further, there is a lot of publicly available human demonstrations from skilled

players. Counter-Strike news website HLTV has decades of professional gameplay logs available for download [50]. We use those demonstrations to curate our 123-hour dataset. Finally, it is possible to modify the game [100]. We rely on this modification capability to implement an agent control interface and to debug our dataset curation system.

1.3 Dissertation Roadmap

Chapter 2 discusses MLMove in relation to prior works on agents for games including efficient hand-crafted agents, superhuman reinforcement learning agents, human-like imitation learning agents, and LLMs.

Chapter 3 provides an overview of the problem domain. This section describes the game, formalizes the human-like agent task, and defines the principles of human-like movement.

Chapter 4 defines the dataset curation system, CSKnow, which both: (a) creates the training data for the learned movement model by tokenizing recorded game state and (b) tokenizes the live game state into model inputs during model deployment.

Chapter 5 defines the learned movement model, including the model input, output, architecture, and training.

Chapter 6 explains the complete agent system, MLMove, that combines the learned movement model with a hand-crafted behavior tree for non-learned behaviors. This section also explains how we use behavior trees to test the agent system.

Chapter 7 evaluates the behavior generated by MLMove relative to humans and two industry baselines. This section demonstrates that MLMove generates more human-like movement and key game outcomes than the baselines while satisfying the agent performance constraints of a FPS. It also demonstrates the importance of key design decisions using ablations. Finally, the section discusses limitations of MLMove.

Chapter 8 specifies the Counter-Strike API we created to train, test, and deploy our human-like agent. We propose that games implement similar APIs to enable future agent development in academia and industry.

Chapter 9 discusses the implications of, how future agents can build on, and future applications of MLMove.

Chapter 2

Related Work on Game Agents

Human-like agents are a long-standing goal of researchers and industry practitioners in fields ranging from video games to robotics, social science, and economics. Video game developers want human-like agents that can play with and against humans [79]. Human-like robots can collaborate with humans and accomplish everyday tasks like rearranging a scene [63, 82]. Virtual towns of human-like agents can enable social scientists to study how people interact [77, 78]. Economists can study crises like the 2008 housing market bubble using virtual economies controlled by human-like agents [89].

The specific sub-problem of human-like navigation is an important component of multiple applications including robotics, autonomous driving, visual effects, and games. For example, crowd simulation in games and visual effects endeavor to generate trajectories for hundreds or thousands of simple agents with much less complex inter-agent interactions than FPS games [75, 88, 109]; while embodied agent motion planning research for robot navigation requires orders of magnitude more compute resources than FPS games to interact with a real physical world observed through cameras [29, 53, 114].

Our work addresses the broader challenge of human-like groups of agents, and the narrower subproblem of human-like motion control, in FPS games. These agents perform a wide range of movements (walk, run, jump) in a dynamic environment under extreme runtime performance constraints. There are three main ways to design agents. Each of these approaches has not achieved significant industry adoption because they fail to address at least one of the three main challenges facing human-like agents: complexity of human behavior, moving in a human-like style, and compute efficiency.

1. Hand-Crafted Rules (Section 2.1) - As explained in Chapter 1, human-like behavior is far too complex to manually encode as a set of rules.

2. Learn to Maximize Objective (Section 2.2.1) - While these agents address the complexity challenge, they learn to maximize an objective, typically win rate, using an inhuman style.

3. Learn to Imitate Human Demonstrations (Section 2.2.2) - While these agents address the complexity and human-style challenges, current approaches are too computationally inefficient to deploy in multiplayer FPS games like Counter-Strike. Computational complexity is not a fundamental limitation of imitation-based approaches. Rather, prior applications of the approach focused on other domains where efficiency requirements are not as strict.

Recently, large language models (and other types of foundation models) (Section 2.3) have gained popularity as a way to design agents. Foundation models are large models trained on general datasets of human language, video, and images before being applied to a specific task. Large language models are a type of foundation model that focus on language. In the agent domain, foundation model approaches are typically extensions of one of the prior approaches. For example, foundation models may automatically write the behavior rules for a rule-based agent, alleviating the difficulty of manually specifying all behaviors. Alternatively, foundation models may write the objective function that agents learn to maximize. Despite these improvements, foundation model-based agents have not achieved industry adoption for the same reasons as the prior approaches: they do not address all three key challenges.

In this section, we explain the prior work on designing agents using each approach, the degree to which each approach has been adopted by the wider game industry, and why prior applications of each approach fail to satisfy all three human-like agent design challenges. Finally, we end with an analysis of other multi-agent applications and how their techniques may apply to our current problem.

2.1 Hand-Crafted Rules

The industry-standard approach for designing human-like agents in games is hand-crafted rules specifying movement and all other behavior. While these agents are efficient, they are not human-like. It is extremely difficult for game developers to hand-craft rules for every possible situation. This limitation has resulted in widespread skepticism by industry developers towards agents in multiplayer FPS games. Counter-Strike removed agents from their competitive experience because the agents' inhuman logic led to abusive experiences [99]. Agents in other multiplayer FPS games, like Fortnite, are only used as primitive training tools for new players while they learn the game's basics [101]. Academic research on hand-crafted agents also demonstrates the struggle to generate human-like behavior in complex environments. Huang et al. require a complex hierarchy just for coordinating movement of pedestrians through doorways [54]. These attempts to use agents demonstrate their potential if they behaved in a human-like fashion.

Due to this skepticism, the game developer community has focused on developing agents for game genres other than multiplayer FPS. One example is singleplayer FPS, where one player faces hoards of agents that do not need to appear human-like. In early singleplayer FPS titles like Doom

(1993), developers wrote flat finite-state machines (FSMs) to control enemy behavior [103]. These basic FSMs were fine for expressing simple logic, but not for the complex logic necessary to generate human-like movement.

Over time, developers created tools for expressing more complex agent behaviors as hierarchies of rules. While these rule hierarchies are not complex enough to generate human-like behavior, they are sufficient to create engaging experiences where one human fights against overwhelming numbers of simpler enemies. Isla proposed a hierarchical approach known as Behavior Trees (BT) for Halo 2 (2004). BTs enable composing complex FSMs out of simpler ones with temporal semantics [55]. BTs are now a standard programming model in the video game industry, with support by major engines like the Unreal Engine [30], as well as the robotics industry [15].

In addition to BTs, two other approaches are popular for organizing hierarchies of rules in videos games. Utility AI-based agents in games like The Sims (2000) use developer-specified utility functions to select the best option given the current game state [84]. Planning algorithms use automated search over developer-specified constraints to find a chain of actions that accomplishes a goal. Orkin proposed Goal Oriented Action Planning (GOAP) for F.E.A.R (2004). In GOAP, developers specify action costs and use A* to find the cheapest sequence of actions to accomplish a goal [74]. Unfortunately, GOAP's A* search can be hard to control and too inefficient. Developers for titles like Transformers: Fall of Cybertron use Hierarchical Task Networks, which give developers greater control by enabling them to manually specify the set of possible plans [84]. While these approaches improve the process of designing complex FSMs, they do not change the fundamental limitation: designers must manually design rules for all of humans' complex behaviors.

2.2 Learning

The game industry recognizes the limitations of hand-crafted rules, and is exploring learning-based approaches to create human-like agents. The two main approaches are training an agent to learn how to maximize an objective, known as Reinforcement Learning (RL), and training an agent to imitate human demonstrations, known as Imitation Learning (IL). Game developers like Electronic Arts and Riot have explored RL-based agents [9, 45], with Electronic Arts comparing them to IL-based approaches [45, 95]. However, the game industry is currently in the early stages of adopting learned agents. No game developer has released a human-like agent for a multiplayer FPS at this time.

2.2.1 Learning to Maximize Objective

RL agents learn to maximize an objective function, typically win rate. The RL agent creation process has two steps: a developer writes an objective function that typically encourages winning, and the agents learn to maximize that function by playing the game. RL agents can generate superhuman

Figure 2.1: An objective function written in Little Learning Machines's visual programming language.

behavior in complex strategy games like Dota 2 and Go [4, 98] and in FPS games like Doom and Quake [57, 62]. The RL approach avoids the limitations of hand-crafted rules, since the learning procedure handles creating a complex behavior policy.

The problem with RL agents is that winning behavior is not human-like and thus not engaging for users. RL agents typically learn to maximize the objective function through self-play, and do not learn from examples of human play. Therefore, they are not likely to generate behavior that matches human expectations. Humans may struggle to collaborate with the RL agents whose actions do not match human expectations [32]. In the racing game MotoGP 20, game reviewers found that the RL agents won with an inhuman level of aggression and disregard for personal safety [36].

Game developers have successfully utilized RL agents in game genres that do not require human-like behavior. In Little Learning Machines, the gameplay revolves around players training RL robots that collect coins and dodge obstacles. Players utilize a visual programming language for writing reward functions. Figure 2.1 demonstrates the reward programming language. Typically, these reward functions return a positive value for actions that lead to winning, such as collecting coins, and a negative value for actions that lead to losing, such as jumping on flowers [8]. Rewarding human-like actions would be challenging, so the developers have been careful to design a game where the agents can be engaging without being human.

A secondary limitation of RL agents is training cost. RL agents need to play many games in order to learn complex policies. Running the game can be computationally expensive, since the engines are designed for human-play rather than simulation. Researchers have explored techniques to train RL agents more efficiently in simulators designed for high throughput [96] or in world models that emulate the game [42]. Developers at Riot wrote a simulator of Teamfight Tactics (TFT) in order to train their RL agent [9]. However, both of these approaches have limitations. The simulator approach requires rewriting the game's logic. Rewriting may be cost prohibitive, and differences

between the game and simulator implementations may prevent policies from succeeding in the real game. Learned world models have had success in subsets of Doom [42] and MineCraft [44], but have not yet been demonstrated for team-based FPS games.

2.2.2 Learning to Imitate

IL agents learn to imitate human behavior. The IL agent creation process has two steps: a developer creates a dataset of human demonstrations, and the agents learn to imitate the demonstrations. When trained on large, diverse datasets sets of human play, IL-based agents can generate human-like movement for a wide range of situations. While game developers like EA have explored IL-approaches [45, 95], no human-like IL agents are currently deployed in multiplayer FPS at this time. Current IL agents are too computationally inefficient for multiplayer FPS games.

IL agents are used to predict human movement around autonomous vehicles, which have different compute constraints. Scene Transformer [73] trained a transformer for predicting multiple pedestrians' and cars' trajectories on different roads over a five-second time horizon. Scene Transformer leverages the transformer's attention mechanism to learn relationships between cars, pedestrians, and road geometry. MotionLM [93] demonstrated that a decoder-only transformer architecture can increase accuracy, since the decoder enforces causal relationships between earlier and later time steps. The models used in Scene Transformer and MotionLM cannot be directly applied to motion control for FPS games, because their compute cost is multiple orders of magnitude too high. They use large model architectures because they target a different use case: predicting pedestrians over a five second time horizon. Adapt [2], a compute-optimized movement model based on the Scene Transformer, runs in 11 ms when highly optimized for a Tesla T4 GPU. Its compute cost is still at least two orders of magnitude greater than the AI budget of FPS games [18]. In contrast, our learned movement model, designed for learning human-like movement in team-based FPS games, requires two orders of magnitude less compute than Adapt without any hardware specific optimization.

One group of researchers have created IL-based human-like agents for multiplayer FPS. Pearce and Zhu trained a model that controls all behavior (not just movement) of a single Counter-Strike agent using rendered images as input [79]. This pixels-to-actions approach (similar to [41, 59]) requires a GPU for every agent, approximately three orders of magnitude (800×) higher compute than commercial FPS games' agent performance constraints. The GPU is necessary so the game can render the frame and the model can process it. Additionally, Pearce and Zhu do not generate coordinated team behavior because they train on data from, and test their agents in, a game mode where players typically practice low-level mechanics without the need for intra-team coordination.

Efficient IL agents have been deployed in commercial games outside of the multiplayer FPS genre. The key challenge is choosing an efficient state representation. Other genres have simpler game states than multiplayer FPS, so game designers can more easily create efficient IL agents. The first Forza game, a racing title, used IL agents on an Xbox console with a 733MHz CPU [102]. The

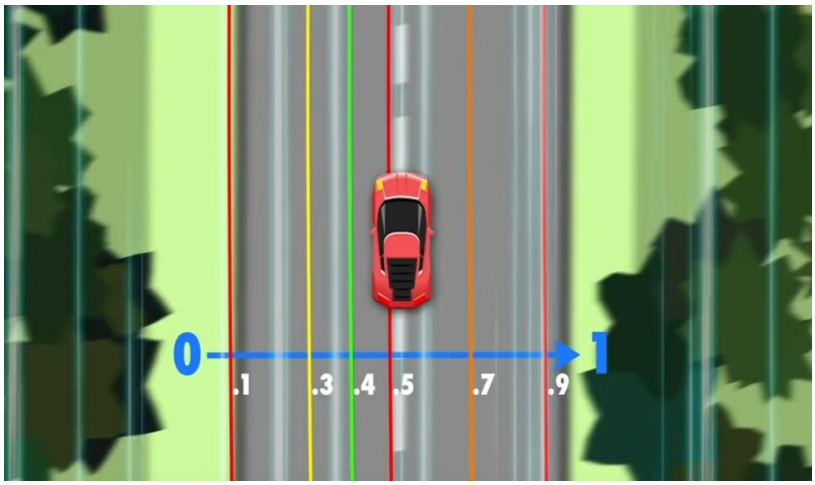


Figure 2.2: Driving game state can be represented as following a 2D path [39].

2D fighting game Killer Instinct used IL agents as well [39]. Racing games and 2D fighting games can be simplified to navigating 2D spaces, as demonstrated by Figure 2.2. Navigating a 2D space without teammates is much simpler than the team-based, 3D movement and positioning necessary in a multiplayer FPS. A key idea of this dissertation is picking an efficient state representation for multiplayer FPS games, thus enabling efficient human-like agents.

Other games have used IL to handle low-level human-like animations. In this application, multiple papers focus on the key idea of using IL to learn a state space of human-like animations, and then learning a separate policy to search that space. Learned Motion Matching searches the space using an IL-trained controller, and Motion VAEs uses a RL-based controller [52, 67]. While these approaches focus on a lower-level form of behavior than our learned movement model, the separation of state representation and state search controller as a way to ensure temporal coherence may generalize across applications.

2.2.3 Reinforcement Learning/Imitation Learning Hybrid

Research on hybrid RL and IL training procedures created objective-based approaches that also generate human-like behavior. GREIL is an RL-based crowd control policy trained with an objective function based on similarity to human examples [11]. Cicero is an agent trained with piKL, which regularizes the objective function with an IL policy to prevent drastic deviation from human behavior. Cicero is designed for Diplomacy, a turn-based strategy game where action frequency is 100 times slower than in a FPS [32]. We are not aware of a hybrid RL/IL approach for human-like agents in a FPS game.

2.3 Large Language Model Controllers

Large language models (LLM) enable improved versions of the previously described rule-based and learning approaches. The game industry is beginning to explore these LLM-based improvements. Google and Square Enix have demonstrated experimental text games with LLM-based agents that imitate human behavior [34, 71]. However, game developers have not deployed these LLM-based improvements. In this section, we will explain how LLMs can enhance each approach for designing agents and the remaining challenges.

LLMs can write code, so they can replace humans for writing rule-based agents. The resulting LLM-crafted rules are efficient, even if the LLM itself is large. However, there are no LLM-crafted rule-based agents that currently are complex enough to generate human-like behavior in a team-based FPS. Code-as-Policies (CaP) uses LLMs to write rules for real-world robots organizing objects on a desk [65]. Voyager improves on CaP by incorporating LLM-written rules in a hierarchical agent that plays Minecraft. This hierarchy consists of (1) using the LLM to write a library of behavior subroutines, and (2) using the LLM to query the library for the appropriate skill in the current game state [106]. LLMs can write better behavior rules by leveraging prompting techniques including Chain-of-Thought, which encourages the LLM to output intermediate reasoning tokens [107], and Chain-of-Code, which enables the LLM to emulate psuedocode subroutines [64]. Startups like Agentic.ai are exploring similar approaches for FPS games [17]. However, an open question remains: can one express a human-like policy using a collection of rules? Even an LLM may struggle to write rules for how humans should coordinate and compete in every possible FPS game state.

Just like humans, LLMs can create objective functions for RL-based agents that learn to maximize the objective. MineDojo leverages LLMs to create reward signals from Minecraft video demonstrations [33]. Eureka enables dexterous robot agents by using an LLM to evolve reward function code [69]. While these RL approaches can solve a general set of small tasks, they are not yet able to generate human-like behavior for FPS games. Additionally, LLM-based RL agents have not been demonstrated in complex games like Dota 2, unlike RL agents using hand-crafted reward functions [4].

In order to generate human-like behavior, LLMs can be utilized to produce high-level plans for IL agents. Park et al. surrounded GPT-3.5 with memory and query systems in order to create human-like daily plans for characters in a 2D world [78]. MindAgent uses a similar approach with GPT-4 to collaborate with humans in CuisineWorld (a 3D version of Overcooked) and Minecraft [38]. CrowdMoGen uses GPT-4 to enable generalizable crowd movement models, as the LLM outputs a situation-specific plan that guides diffusion model for human movement [40]. While these general approaches can generate human-like behavior in a wide range of situations, they all rely on GPT models that are orders of magnitude too expensive for the performance constraints of FPS agents. Even GPT-3 requires requires 32,000 times more parameters than our learned model [5].

2.4 Other Multi-Agent Applications

There are a wide range of multi-agent applications from navigating warehouses [13, 76] to controlling multiple units in a real-time strategy game [92] and competing in a Sumo wrestling game [6]. While these applications typically require the team of agents to "win" at an objective rather than imitate humans, the RL solutions still have valuable insights that may be relevant for human-like agents in multiplayer FPS. The key insights in these applications are modeling other agents behaviors and the world. Modeling opponent behavior makes them easier to predict and respond to [6], and modeling teammate behavior improves coordination [13, 68]. Similar to these cross-agent models, we use transformer's attention mechanism to learn relationships between players. Modeling the world enables agents to better predict and respond to changes in the world. More accurate and complex world simulations require more complex agent coordination [92]. Learned world models [43, 113] and faster hand-crafted world simulators [60] enable agents to learn the world's dynamics at a faster rate. In this dissertation, we implicitly learn the world (map geometry) from agents' positions. Future work on human-like agents may explore more advanced world models in order to better generalize to multiple maps.

Chapter 3

Task Definition: Human-Like Multiplayer FPS Agent

This dissertation defines a system for creating and evaluating a human-like agent in Counter-Strike. This chapter describes the human-like agent task: the details of human-like behavior in Counter-Strike and how an agent interfaces with the game. While we focus on one game, games across the multiplayer FPS genre share our task definition since they have many similar key properties, like the importance of team-based movement. We define the human-like agent task in three steps:

1. **Game of Counter-Strike** (Section 3.1) - define the game including the map geometry and players' objectives. In order to explain our system, we must first explain the world surrounding the agent. Humans cooperate and compete in order to accomplish their objectives. The objectives and strategies to accomplish the objectives are the motivation for humans' behaviors, like movement.

2. **Agent Task Formalization** (Section 3.2) - define what an agent does including the input observation space, output action space, and environment state space. The second step describes how an agent (a) perceives the environment and (b) emits actions that generate the desired behavior. We will also provide a high-level description of our agent's hybrid architecture combining learned and hand-crafted components. Later chapters will describe the details of the individual components.

3. **Principles of Movement** (Section 3.3) - define the desired characteristics of a human-like agent's behavior. We focus on movement since it is the key human-like agent subtask, both difficult to generate and crucial for skilled human play.

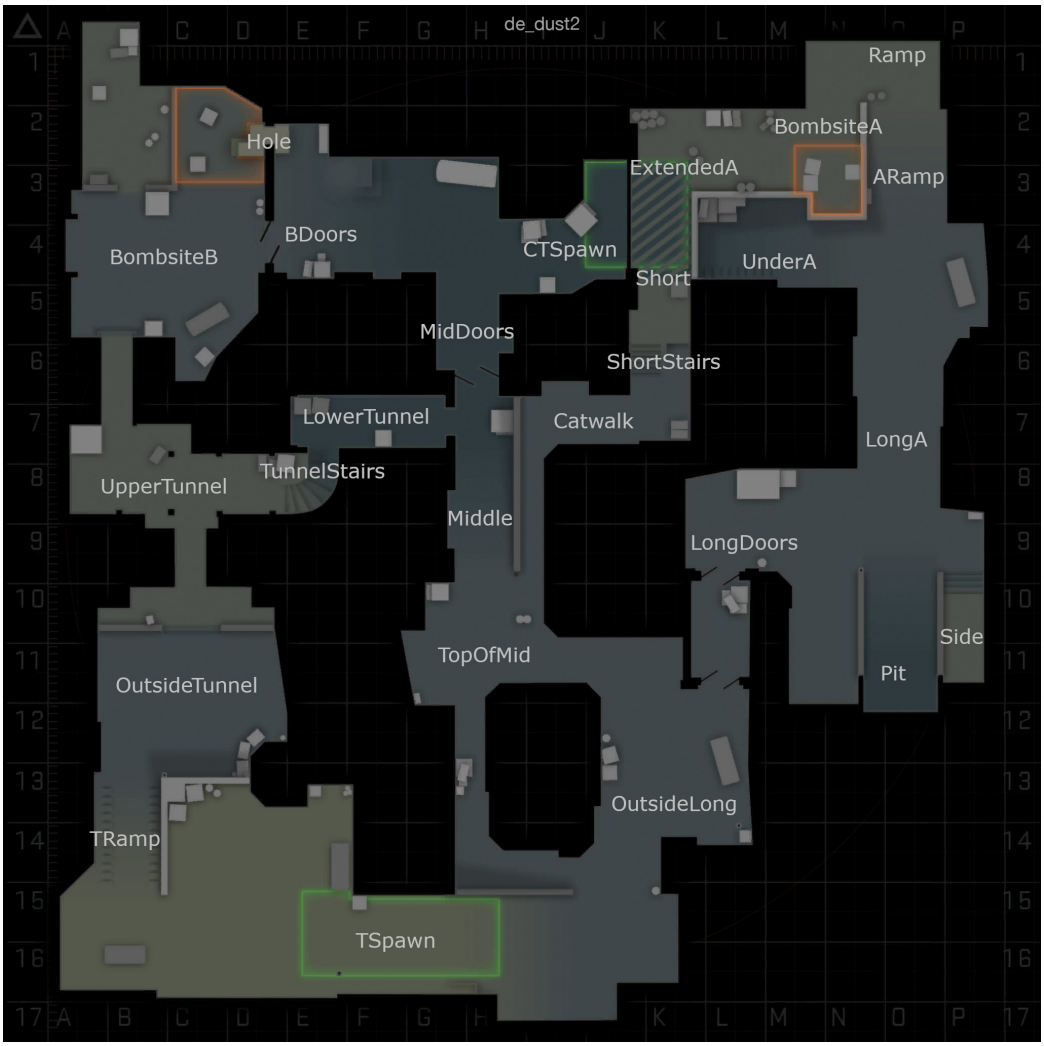


Figure 3.1: Our agent focuses on the map de_dust2. After several decades of play, humans have labeled the important regions of the map in order to coordinate team-based movement strategies. The bombsites are the regions outlined in red in the top right and left of the image. de_dust2 is a canonical example of Counter-Strike maps, and all other maps used for competitive play have similar labels.

Figure 3.2: The defense (yellow outline) and offense (blue outline) players in Counter-Strike's retakes game mode.

Figure 3.3: In Counter-Strike's retakes game mode, the offense (blue outline) attempts to defuse the bomb and the defense (yellow outline) attempts to prevent the defusal.

3.1 Game of Counter-Strike

Counter-Strike is a multiplayer FPS involving two teams competing for control over a map. To focus on player movement, we concentrate our attention on a popular Counter-Strike practice mode known as "Retakes." In each round, a bomb is planted in one of two pre-determined regions known as bombsites A and B. Figure 3.1 shows the locations of the bombsites on one popular map called de_dust2. The bomb will explode in 40 seconds unless it is defused. The goal of one team, who we call the defense, is to defend the bomb until it explodes. At most 3 players are on defense. The goal of the other team, who we call the offense, is to defuse the bomb before it explodes. At most 4 players are on offense. As demonstrated in Figure 3.2, we highlight the defense players with a yellow outline and the offense players with a blue outline for visual clarity. Figure 3.3 demonstrates the offense defusing the bomb and the defense preventing the defusal. One defense player must start at the bomb location while all other players can start at any location on the map. Members of the two teams can eliminate each other using several weapons and grenades. To further our focus on movement behavior, we restrict all players to the same weapon type and preclude the use of any grenades.

Our focus on retakes reduces the amount of Counter-Strike necessary to model. Other aspects of the game state include cross-round state like the number of rounds that each team has won in the match and the amount of money each player has to buy weapons. We simplify our game state model by focusing on a practice mode in which all rounds are independent.

We also simplify our state model by training agents for a single map, de_dust2. Even though FPS games like Counter-Strike can have many maps, maps are designed to have similar room and path layouts that are known to enable interesting gameplay; expert players tend to hone their strategies by playing on the same map over and over [97]. For these reasons, we chose to focus our study on the extremely popular de_dust2.

3.2 Agent Task Formalization

3.2.1 Game State

We model the subset of Counter-Strike's state necessary to represent retakes. The game state at time t consists of player states $q_{i,t} \in Q_t$ as well as global game state that consists of the map state map_t and key events $e_{i,t} \in E_t$ like players shooting or being eliminated. Time t is tracked inside each round of Counter-Strike using game ticks. For the rest of this dissertation, we use game ticks (steps) and time t interchangeably. We use $\mathbb{B}$ to represent $\{\text{True}, \text{False}\}$ and $\mathbb{Z}$ to represent the set of all integers.

1. Each player's state $q_{i,t} = [p_{i,t}, v_{i,t}, u_i, l_{i,t}, vd_{i,t}, h_{i,t}, r_{i,t}]$ consists of position $p_{i,t} \in \mathbb{R}^3$, velocity $v_{i,t} \in \mathbb{R}^3$, team $u_i \in \{\text{Offense}, \text{Defense}\}$, alive status $l_{i,t} \in \mathbb{B}$, view direction $vd_{i,t} \in \mathbb{R}^2$, health $h_{i,t} \in \mathbb{Z}$, and armor $r_{i,t} \in \mathbb{Z}$.

2. Map state $map_t = [b_t, x_t]$ consists of the target bombsite $b \in \{\text{A}, \text{B}\}$ and seconds left until the bomb explodes $x_t \in \mathbb{R}$.

3. Each game event $e_{i,t} = [src_{i,t}, tgt_{i,t}, y_{i,t}]$ consists of source player id $src_{i,t} \in \mathbb{Z}$, optional target player id $tgt_{i,t} \in \mathbb{Z}$, and type $y_{i,t} \in \{\text{shoot}, \text{hurt}, \text{elimination}\}$.

3.2.2 Action Space

Players can move, aim, and fire. We refer to these components of a player's action at time step t $a_{i,t} = [m_{i,t}, du_{i,t}, f_{i,t}]$ as movement command $m_{i,t} \in \mathbb{Z}$ specifying which direction to move, how fast, and whether to jump or not; aim command a.k.a view direction update command $du_{i,t} \in \mathbb{R}^2$; and fire command $fc_{i,t} \in \mathbb{B}$.

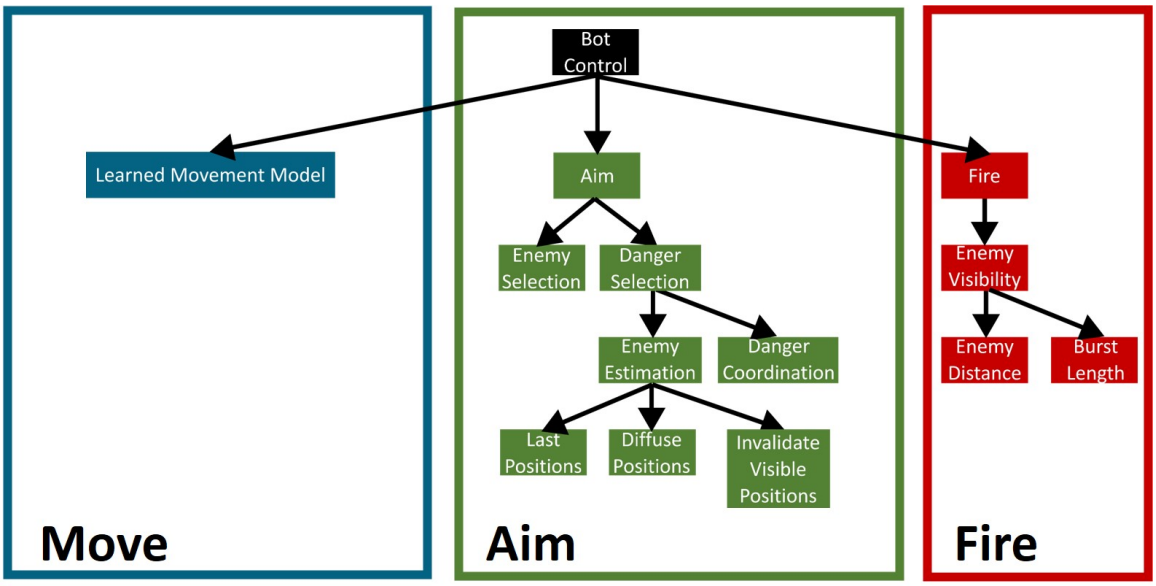


Figure 3.4: MLMove's architecture consists of three components (moving, aiming, and firing) organized in a tree structure.

3.2.3 Agent Architecture

Our goal is to design human-like controllers for moving, aiming and firing. Figure 3.4 shows the architecture of our agent, known as MLMove. We designed MLMove's architecture as a tree with branches for moving, aiming, and firing.[1]

Human-like movement is hard and is the focus of this dissertation. We use an imitation learned movement model in order to generate behavior that demonstrates humans' complex movement and positioning principles. Chapter 5 describes the learned movement model.

We use hand-crafted modules for aiming and firing. There is a large body of work on human-like aiming and firing [3, 56, 70] since these behaviors are easier to encode with rules. Players aim at enemies that are visible, or at locations where enemies are likely to become visible. Similarly, players fire when they are aiming at enemies. Chapter 6 describes the hand-crafted aiming and firing components, as well as the overall structure combining the components with the learned movement model.

[1]The nodes in Figure 3.4 are a simplification of the actual MLMove nodes. We use this simplified representation for visual clarity.

(a) The blue offense player leverages cover (red wall) to hide from the yellow defense player above.

(b) The blue offense player stands near the opposite wall to engage with the yellow defense player.

Figure 3.5: An example of the cover movement behavior. The yellow and blue lines showing players' view directions are for demonstration purposes, and are not visible when playing the game.

3.3 Principles of Movement

Out of the three components in our architecture, we focus on learning movement because it is the hardest to handle with hand-crafted rules and fundamental to skilled play. In this section we address "what is human-like movement" that we hope to emulate in our agent. Movement and positioning (moving through key areas) principles provide players with an advantage over their opponents. Direct interactions between enemies are brief in Counter-Strike, often lasting only a few seconds, and every bullet is important, since only one hit is necessary to score a kill. The key principles provide advantages including allowing players more time to react than their opponents and making their bullets more likely to hit the target.

Despite their importance, movement and positioning principles are difficult to implement, particularly in industry-standard non-learned agents. The application of the principles depends on a wide range of factors including map geometry, teammate positions, and enemy positions. It is not feasible to hand-craft rules for every possible situation, even when limiting agents to one map like de_dust2. As evidence, we will provide a high quality, hand-crafted agent that a Counter-Strike expert spent months crafting, and this agent's movement was not close to human-like. But, we can quantify the movement behaviors' (and resulting strategies') usage rates using game metadata and expert humans' terminology for describing the strategies. In this subsection, we will explain the principles, provide examples of behaviors implementing the principles, and explain why the principles are difficult to implement with hand-crafted logic. These movement-based strategies are common across the multiplayer FPS genre, so an approach implementing these principles should generalize to many other games.

3.3.1 Principle 1: Cover Usage

The first principle is cover usage. Cover is map geometry that blocks sight lines and bullets between a player and their enemies. Players use cover to stop engagements. Players move into the open, away from cover, when they want to maintain sight lines to an enemy and increase the likelihood of bullets hitting an enemy. This enables players to engage enemies.

Figure 3.5 demonstrates the cover usage principle. In Figure 3.5a, the blue offense player uses a position near cover walls to protect themselves from enemies. The red wall is cover as it prevents the yellow defense player from seeing the blue offense player. When humans are ready to engage with the enemy, they move to other parts of the map. Figure 3.5b shows a blue offense player using a position near the opposite wall, away from cover, so they can see as much of the yellow defender as possible and their bullets have a high probability of hitting the defender. Additionally, humans do not stand in the middle (the red box in Figure 3.5b) because then they can get shot without seeing everyone.

Hand-crafted agents struggle to generate human-like cover utilization. It is not as simple as standing near walls. Cover depends on both map geometry and player positions. A wall only forms cover if it is between a player and an enemy. The blue offense player in Figure 3.5b is standing near multiple walls, but the walls do not provide cover as they do not block the sight line to the enemy. Also, multiple walls may form cover between a player and the enemy. It is not feasible to craft movement rules for every possible combination of player positions and map geometry in de_dust2, let alone across many different maps. When we explored computing cover only from map geometry, we did not find the analysis useful for predicting player behavior because so much of the map geometry serves as possible cover [23, 27]. Human behavior demonstrations identify which walls humans use as cover in each game state.

Heatmaps like Figure 3.6 demonstrate humans' use of cover. The heatmap shows where humans move on the subset of de_dust2 from Figure 3.5. Each pixel counts the game ticks when an offense player occupies that position in the map. Red indicates humans use that position more frequently. The red at the top of the heatmap shows that humans stand near the right wall from Figure 3.5a, which provides cover. The red at the bottom of the heatmap shows that humans also stand near the left wall from Figure 3.5b, which is in the open and allows them to engage enemies. They do not stand in the blue region in the middle, which is neither behind cover nor completely in the open. Here, players will struggle to fight back against enemies during an engagement.

3.3.2 Principle 2: Reacting To Enemies

The second principle is reacting to enemies. Humans adjust their movement and positioning based on enemy locations. If the offense is about to defuse the bomb, then the defense must move to positions near the bombsite to stop the defusal. If the offense is far from the bombsite, then the defense should try to intercept the offense in cover-free positions before they reach the bombsite.

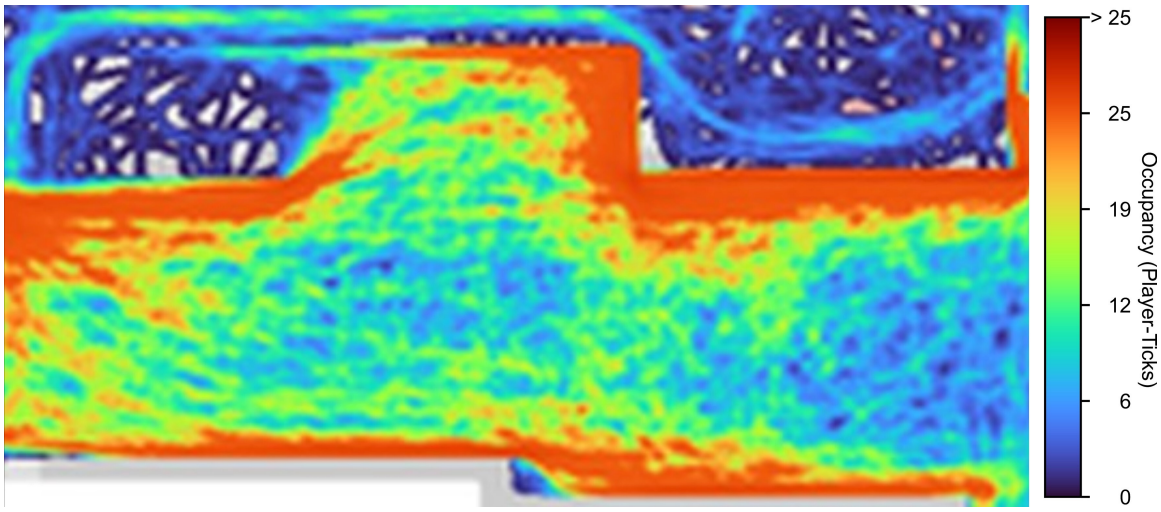


Figure 3.6: A measure of cover usage from human gameplay. Humans use red positions more frequently. The red regions in the top and bottom show that humans use cover to hide from enemies and then engage the enemies by standing in the open.

(a) The yellow defense player moves to the left to stop the blue offense player from defusing the bomb.

(b) The yellow defense player moves to the right to stop the blue offense player while they climb the stairs.

Figure 3.7: An example of the players' changing their movement behavior in response to enemy positioning.

Similarly, the offense can move faster and with less concern for cover if the defense is far away. If the defense is near, the offense needs to move slowly and cautiously without straying far from cover. The appropriate movement and positioning behavior depends on all players positions.

Hand-crafted agents struggle to react to enemy positions in a human-like fashion. Just like with cover-based movement, this behavior is situation specific and it is not possible to craft rules for every situation. For our hand-crafted baseline agent, we spent months manually annotating how our hand-crafted, baseline agent should respond to enemy positions. For example, we manually annotated defense positions far from the bombsite known as chokepoints, defense positions closer to the bombsite, and rules for how defense agents should pick between these positions. Chokepoints

provide the defense with an advantage since they are narrow, constrained regions where the offense is far from cover. But, we could not write down rules that always made every judgment call in a human-like fashion. If a defense player is assigned to a chokepoint, they eventually need to make a judgment call to deprioritize the chokepoint and move to the bombsite because no offense player is likely to move through the chokepoint. We were not able to craft rules that always chose "chokepoint or bombsite" in a human-like fashion. Demonstrations are necessary to learn how a human would make such a judgment call.

Figure 3.7 provides an example of reacting to enemies. In Figure 3.7a, the yellow defense player moves to the left (BombsiteB) in order to stop the blue offense player. The offense player is defusing the bomb, so the defense player needs to prioritize the bombsite. If the defense player does not react, then the defusal will be successful and the defense player will lose the round. In Figure 3.7b, the offense player is coming up the stairs on the right rather than standing in the bombsite. The defense player should respond by moving to the right to intercept the offense player on the stairs, a chokepoint. If the defense player moved to the left, they would needlessly yield control over a portion of the map and allow the offense player to travel through the chokepoint without trouble. Thus, the defense player maximizes their odds of wining by reacting to the offense player's position.

3.3.3 Principle 3: Flanking

The third movement principle is flanking. Flanking is synchronized attacks from multiple directions. Defense players naturally have an advantage, since they can set traps at chokepoints and wait for offense players. At these chokepoints, defense players can improve their reactions by predictively aiming where the offense will appear. Flanking enables skilled offense players to gain the advantage by creating moments when the defense is surprised and overwhelmed. The defense cannot predictively aim if they do not know where the offense will appear and must watch multiple directions simultaneously. The difficulty with flanking is determining where to synchronize attacks. Depending on defense player positioning, different offense players in a flank may need to move faster or slower in order to ensure all players enter combat simultaneously.

Figure 3.8 demonstrates this behavior. The defense is in BombsiteB, the bottom right of the image, in order to defend the bomb. The offense is entering BombsiteB to defuse the bomb. In order to execute a flank, the yellow offense players enter from multiple directions: UpperTunnel (red box, top right), BDoors (blue box, middle left), and Hole (green box, bottom left). (Region labels are defined in Figure 3.1.) The offense's flank forces the blue defense players on BombsiteB to watch multiple directions simultaneously. The defense does not know where the offense will come from, so the defense cannot predictively aim at one entrance such as the red one.

Figure 3.8 also demonstrates flanking's temporal synchronization challenges. The offense player using the red entrance may see enemies before they reach the end of the tunnel. In this case, the offense players using the blue and green entrances will need to enter BombsiteB sooner.

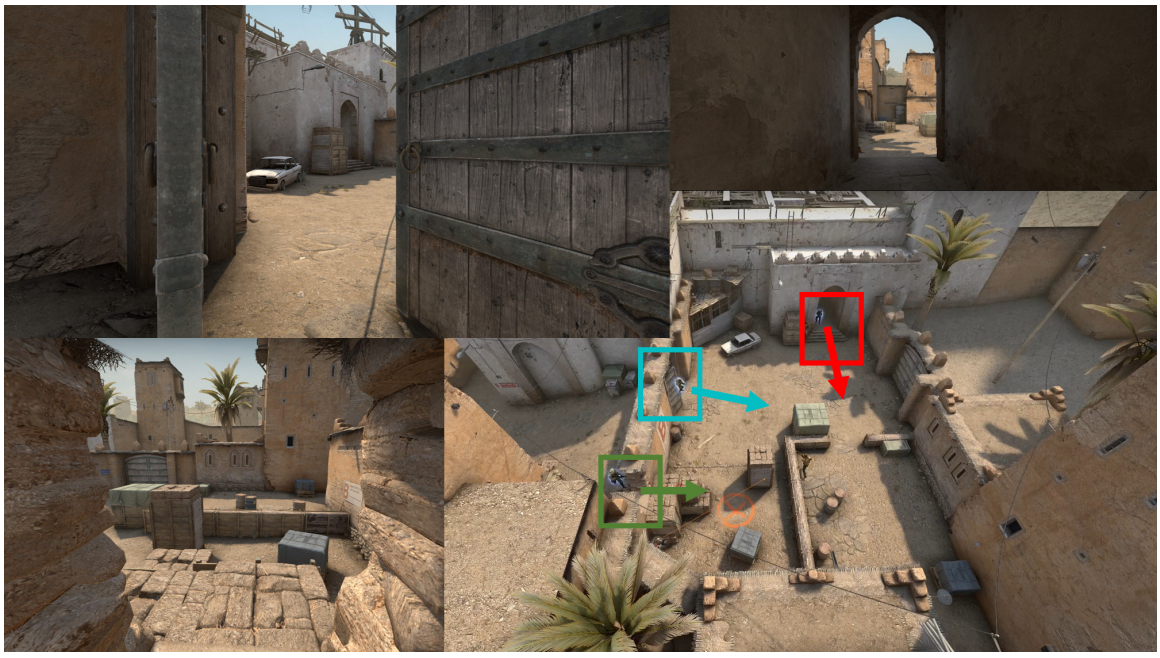


Figure 3.8: Our agent executing a flanking strategy. A learned movement approach enables our agent to generate temporally synchronized, human-like movement that is difficult to encode with hand-crafted rules.

Hand-crafted agents struggle to execute human-like flanking strategies due to the difficulty of adjusting temporal synchronization for enemy positioning. For our hand-crafted baseline agent, we spent months manually annotating map positions where offense agents should wait while temporally synchronizing with their teammates. However, it was difficult to write logic for how to adjust these synchronization positions depending on enemy positions. It was not feasible to adjust the synchronization logic for the long tail of possible enemy locations. As a result, players found our hand-crafted agents always flanked in the same way and were predictable.

Chapter 4

CSKnow Dataset Curation System

Our hybrid agent uses both rule-based and learned components. Due to the complexity of human movement, we use a large dataset of human demonstrations to train an IL movement model. No suitable datasets existed when we started this project, so we had to curate the first large scale dataset for learning team-based movement in a popular FPS game featuring professional players. In this chapter, we present our dataset curation system called CSKnow.[1] We will explain (1) how we used CSKnow to extract the dataset from Counter-Strike gameplay logs, and (2) how we derived a number of high-level features useful for learning FPS movement.

We record game state at a high frequency. Our model makes a prediction every 125 ms to match human reaction times [7]. To enable our model to learn these fine-grained behaviors, we extract our features at 16 Hz frequency (every 62.5 ms). Prior Counter-Strike datasets focused on long-term outcomes like win probability, so they captured game state at too low frequency for evaluating movement commands at every 125 ms. For example, ESTA contains professional game play with data points every 500 ms [110], and PureSkill.GG contains amateur game play with no guarantees on data capture frequency or even if some data were dropped [19].

The derived features address two limitations. First, we add derived features describing key events like shots and kills. Players communicate these key events to teammates during gameplay. Counter-Strike logs do not record this communication, so we add derived features recording recent key events and indicating key events may occur soon. Second, humans behavior is guided by long-term intent. We train our model using a short-term loss function that may not capture these latent strategies. Our derived strategy feature infers long-term behavior.

When designing the dataset, we chose a symbolic state representation instead of rendered images. Games servers and logs already use a symbolic state representation, so we can extract it (with a lot of technical effort). The symbolic representation is far more efficient than rendered images used in other Counter-Strike agent research [79]. Rendered image state requires GPUs both to run the

[1] For the rest of this dissertation, we will use CSKnow to describe both the system and the dataset.

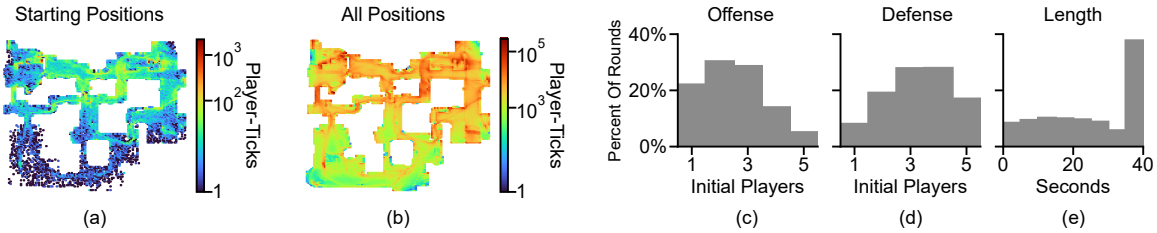


Figure 4.1: CSKnow is a diverse dataset of 123 hours of professional Counter-Strike play. (a) Density of player positions at the start of each round. Players start in a wide range of positions. (b) Density of player positions throughout the entire round. Players visit all areas of the map. (c)-(d) Rounds start with different numbers of offense and defense players, and can end almost immediately or last until the explosion (e). Note: (a)-(b) graphs are log scale, (c)-(e) graphs are linear scale.

game's rendering pipeline and to run large models performing perception. A symbolic representation enables CPU-only agents, since the features can be extracted from the memory of a CPU-based game server and processed by a much smaller model.

4.1 Methodology

We created a system to curate the 123 hour dataset from logs of 1156 hours played on the de_dust2 map by professional players between April 2021 and November 2022. The data comes from over 17K rounds and features 2292 unique players, 513K shots, and 29K eliminations. We downloaded the logs from HLTV [50]. The logs contain game play from the complete Counter-Strike game mode, not just the retakes practice one. Unlike the retakes mode, the complete game mode requires five on each team at the start of each round and involves an earlier stage where teams compete to plant the bomb. We filter the data in CSKnow to game ticks when the bomb has been planted and at least one player is alive on both teams, a super-set of retakes. This filter ensures our dataset is focused enough to be representative of the retakes mode play style while still broad enough to cover a diverse range of gameplay.

Figure 4.1 shows that CSKnow covers a diverse range of play situations: players start in a wide variety of starting positions, and over the course of play move into all locations on the map. Since bomb plants in a full game occur in the middle of Counter-Strike rounds, the number of players that are alive on each team at the time of bomb plant varies significantly in the data set.

4.2 Derived Communication Features

A limitation of the raw Counter-Strike log files is that they do not store audio signals and communication between teammates. Skilled Counter-Strike players make decisions based not only on their knowledge of the de_dust2 map (e.g., intuition about where enemies are likely to be), but also

on information they receive while playing. Players track the following team interactions to know where enemies will be and which teammates need help: is a teammate about to shoot an enemy, is an enemy about to shoot a teammate, or was a teammate recently hurt by an enemy's shot? Additionally, teams use communication to pick a coordinated, long-term strategy. We enrich the dataset with auxiliary player features that can approximate this information.

We derive these auxiliary player features $d_{i,t} = [cd_{i,t}, ts_{i,t}, th_{i,t}, vi_{i,t}, ve_{i,t}, s_t]$ using game events and player states. We identify when a player is about to shoot an enemy with $cd_{i,t} \in [0, 1]$, the distance from a player's crosshair to their nearest enemy (up to 30 degrees). We track the time since they last shot (up to five seconds) with $ts_{i,t} \in [0, 1]$, and the time since a player is last hurt (up to five seconds) with $th_{i,t} \in [0, 1]$. We track the time since an enemy was last visible (up to five seconds) with $vi_{i,t} \in [0, 1]$ and the time since a player might be seen by an enemy (up to five seconds) with $ve_{i,t} \in [0, 1]$. We describe strategy s_t below.

Strategy Control. Humans' actions are guided by global, long-term intent. In Chapter 5, we will explain that our model focuses on short-term predictions in order to handle Counter-Strike's rapidly changing game state and to improve model efficiency. We do not expect our model to learn long-term, latent intent from our short-term loss function.

One key strategic decision is whether to win the current game round. All retakes rounds are independent, so players try to win every round, known as pushing. The professionals in the CSKnow dataset play a different game mode with inter-round dependencies. As a result, they sometimes employ a strategy of intentionally losing the current round to improve the odds of winning future rounds, known as saving. We account for the strategy (s_t) mismatch by labeling each data point with whether the players are trying to win the round at that time: $s_t \in \{\text{Push}, \text{Save}\}$.

We create the strategy labels using the following steps.

1. A Counter-Strike expert manually labeled 323 of the 17216 rounds in the dataset (1.9%). Each round is labeled with a continuous value between 0.0 (save-only round) and 1.0 (push-only round), where intermediate values indicate the time of a switch from pushing to saving during the round. For example, 0.25 indicates a push round in the first 25% of its length. This transition is uni-directional. Players may flip from push to save. However, once they opt for saving, they do not have time to re-engage given the short length of retakes rounds.

2. The remaining 98.1% of the data is labeled using a nearest-neighbor classifier: the aggressiveness of unlabeled rounds is determined by finding the most similar round that is labeled and in the training dataset. Computing similarity between two rounds is hard because player trajectories may end early (upon death) and there are many possible permutations of the mapping between players in two rounds. To address these challenges, we introduce Dynamic Time Warping Average Displacement Error (DTWADE) in Section 4.2.1. At a high level, DTWADE first searches along all possible player permutations to find the best match between the two rounds. Then it refines the estimate of the distance by applying Dynamic Time Warping

(DTW) [91] to the players' trajectories to find a monotonic transformation of the time axis that minimizes the distance between the two sets of temporal sequences.

3. Once each round in the dataset has been labeled, we propagate these labels to the ticks in each round. All game ticks receive a binary s_t label based on their temporal position in the round. The per-tick labels are passed as conditioning input to the movement model of MLMove during its training.

Since humans always use a push strategy in retakes, the 1430 test rounds in the evaluation all have a strategy round value of 1.0 (all ticks in the round are labeled Push). The eight rounds in the user study are drawn from these 1430 rounds.

4.2.1 Dynamic Time Warping Average Displacement Error

Metrics to evaluate the distance between two sets of N trajectories of length T, $r_i = \{\boldsymbol{x}_{i,n}(t)\}_{n=0...N-1}$ for $i = [0, 1]$ and $t = [0, ..., T-1]$, have been proposed in the field of social behavior study; for instance, Average Displacement Error (ADE) [1, 80] is the average displacement between the positions of corresponding actors in the two sets:

$$\text{ADE}(r_0, r_1) = \frac{1}{NT} \sum_{n=0}^{N-1} \sum_{t=0}^{T-1} \left|\left| \boldsymbol{x}_{0,n}(t) - \boldsymbol{x}_{1,n}(t) \right|\right|_2 . \tag{4.1}$$

This measure has recently been extended to measure the distance between a ground truth set of trajectories and the ones created by a generative model (see the Joint Average Displacement Error, JADE [108]). Unfortunately, these metrics are unfit for our problem as: (a) they assume the mapping between trajectories, players within the game, in the two sets to be known, and (b) they require all trajectories to be the same length T.

In Counter-Strike, players may be eliminated early (i.e., each trajectory $\boldsymbol{x}_{i,j}$ has a different length $T_{i,j}$). In addition, the mapping between players in different rounds is unknown, therefore computing the distance between r_0 and r_1 requires testing multiple mapping permutations, which calls for a computationally efficient implementation of the metric.

We propose Dynamic Time Warping Average Displacement Error (DTWADE), to address these limitations when computing the distance between two rounds. First, for each round, all the trajectories are extended to the length of the longest trajectory in that round. Recall that a trajectory ends early when a player is eliminated. We assume an eliminated player remains at the same location until the round ends.

Determining the correct mapping between agents requires testing all of the different permutations. To make the computation more efficient, we sample only $K = 11$ equally-distanced points from the trajectory when determining the mapping between players $(0, 0.1, \ldots, 0.9, 1.0)$. In the case where

Table 4.1: Per Tick Features

Name	Type
Log File	String
Game Id	Int
Round Id	Int
Tick Id	Int
Bomb Plant Site	Categorical
Bomb Fuse Time	Float

the n-th trajectory of r_0 maps to the n-th trajectory of r_1, we can then define:

$$\text{KADE}(r_0, r_1) = \frac{1}{NK} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left|\left| \boldsymbol{x}_{0,n}\left(\frac{kT_0}{K-1}\right) - \boldsymbol{x}_{1,n}\left(\frac{kT_1}{K-1}\right) \right|\right|_2 . \tag{4.2}$$

The KADE is used to find the best mapping between players across two rounds. To do so, we denote the q-th permutation of the trajectories in r_1 by $\text{perm}(r, q)$ and compute the optimal permutation as

$$p(r_0, r_1) = \underset{q}{\text{argmin}}\ \text{KADE}(r_0, \text{perm}(r_1, q)) . \tag{4.3}$$

The metric above provides an approximate, yet efficient, way of finding the closest permutation. Once the correct permutation has been selected, we return the distance by stretching and comparing the trajectories through Dynamic Time Warping (DTW) [91], which finds a monotonic transformation that minimizes the distance between two temporal sequences:

$$DTW(r_0, p(r_0, r_1)) , \tag{4.4}$$

where $DTW(r_0, r_1)$ is the temporal alignment between r_0 and r_1 stretched according to the DTW rules.

4.3 Complete Features Listing

There are three sets of features in our dataset. Table 4.1 lists the overall features that are shared across players. Table 4.2 lists the features that are specific to each player. All "prior time steps" features extend 3 seconds into the past at a 125 milliseconds frequency. Table 4.3 lists the features tracking grenades thrown by a player. Grenades require their own set of feature set because they are relatively rare events (each player can throw only a few grenades in a round) and each event has a lot of data (XYZ coordinates for all time steps during which the grenade exists).

Table 4.2: Per Player Features

Name	Type
Player Id	Int
Alive or Dead	Bool
Team	Bool
Health	Float
Armor	Float
Helmet	Bool
Weapon Id	Int
Walking	Bool
Crouching	Bool
View Angle at Current Time Step	Float
Position at Current and Prior Time Steps	Vector of Floats
Velocity at Current Time Step and Prior Time Steps	Vector of Floats
Crosshair Distance to Nearest Enemy (up to 30 Degrees) at Current and Prior Time Steps	Float
Distance to Nearest Enemy and Teammate at Current and Prior Time Steps	Float
Hurt in Last 5s	Float
Fire in Last 5s	Float
Enemy Visible in Last 5s (Without Considering FOV)	Float
Enemy Visible in Last 5s (Considering FOV)	Float
Hurt in Next Tick	Bool
Kill in Next Tick	Bool
Killed in Next Tick	Bool
Fire Weapon on Current Tick	Bool
Decrease Distance to Bomb Over Next 5s/10s/20s	Bool
Movement Commands	Categorical

Table 4.3: Per Grenade Throw Features

Name	Type
Thrower Id	Int
Grenade Type	Int
Throw Tick Id	Int
Active Tick Id	Int
Expired Tick Id	Int
Destroy Tick Id	Int
Positions During Throw Trajectory	List of Vector of Floats

4.4 System Requirements

The main limitation of a symbolic representation is that we had to write the infrastructure for extracting the game state. The game state must be extracted both (a) from logs for training and (b) from the game binary for inference. Counter-Strike does not have standard APIs for interfacing with logs or the game binary. We created custom game modifications to extract the state. Acquiring similar datasets will be critical for future research on human-like agents. We believe that future games can provide APIs for extracting game state. We describe our suggested API requirements for game engines in Chapter 8.

Chapter 5

Learned Movement Controller

Human movement is complicated. Our agent uses a learned movement model to imitate human movement principles like taking cover, reacting to enemies, and flanking. At a high level, the model receives game state as input and predicts all players' next movement commands. Figure 5.1 shows this combination of inputs and outputs: bounding boxes indicate the game state including all players' positions, and arrows indicate their next movement commands. The model's architecture is inspired by previous work on human-like movement [73]. However, we had to make a number of key decisions in order to satisfy our quality and efficiency requirements of generating human-like movement while using only a small fraction of a single CPU core. In this chapter, we will talk about those design decisions and our training procedure.

5.1 Key Decisions

In a FPS game, human players not only have a complex action space, but also demonstrate complex inter-player interaction and coordination that are quite challenging to model in a rule-based system. However, recent work in transformer models show how to imitate the *effect* of complex human decisions and interactions without modeling the *intermediate* steps (or decisions) that led to the final actions.

The architecture of our movement controller (Figure 5.2 shows a simplified version) is inspired by Scene Transformer [73], one of many [2, 93, 112] transformer-based multi-agent motion prediction systems for pedestrians and autonomous vehicles. The Scene Transformer encodes the state of each agent as input tokens and leverages attention to learn the relationships between all the agents. Like Scene Transformer, our movement controller can also benefit from the transformer architecture's ability to capture rich player interactions with the attention mechanism, process players' state in any order due to the permutation invariance of input tokens, and handle eliminated players with attention masking [104].

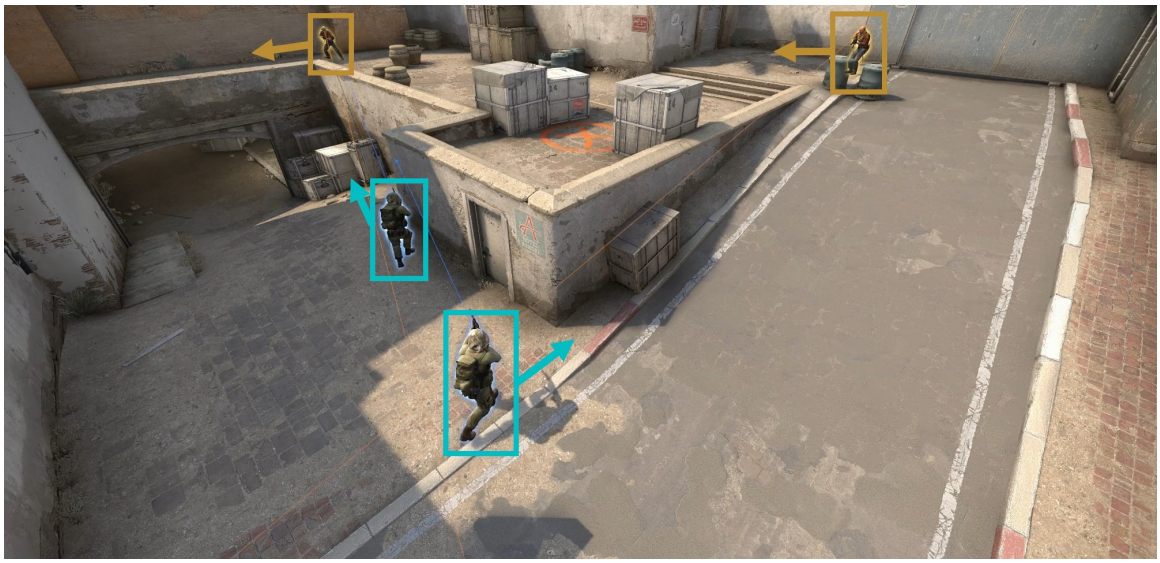


Figure 5.1: The objective of our learned movement model is to efficiently predict all players' movement commands.

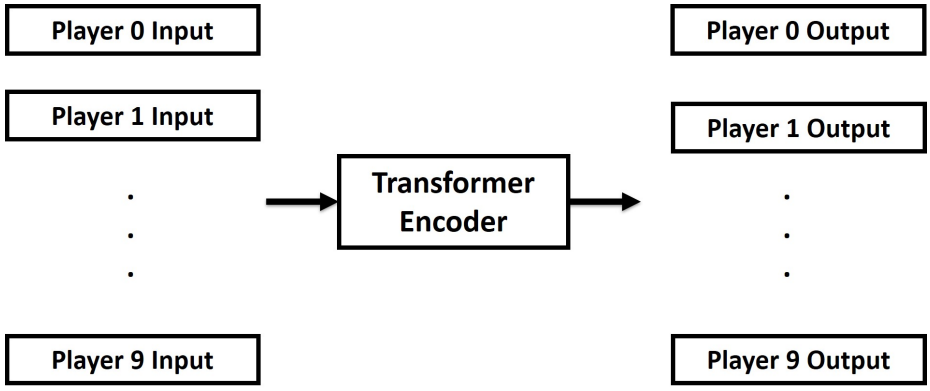


Figure 5.2: A simplified version of our learned movement model. The simplified transformer processes 10 input tokens and outputs 10 movement commands, one per player.

However, Scene Transformer's query latency and compute resources were orders of magnitude higher than what was acceptable for FPS game adoption. We leverage the significant differences between the target applications of FPS games and autonomous vehicles to make architecture and system design choices to create a movement model that: (a) is able to emulate the effect of complex human team play strategy and interactions in a FPS game, and (b) can be executed within the strict compute constraints required by FPS game servers. We highlight five key design decisions below. These application driven design choices enabled us to create a model that can predict human-like movement decisions for two teams of Counter-Strike players (10 total players) within 8 ms per query

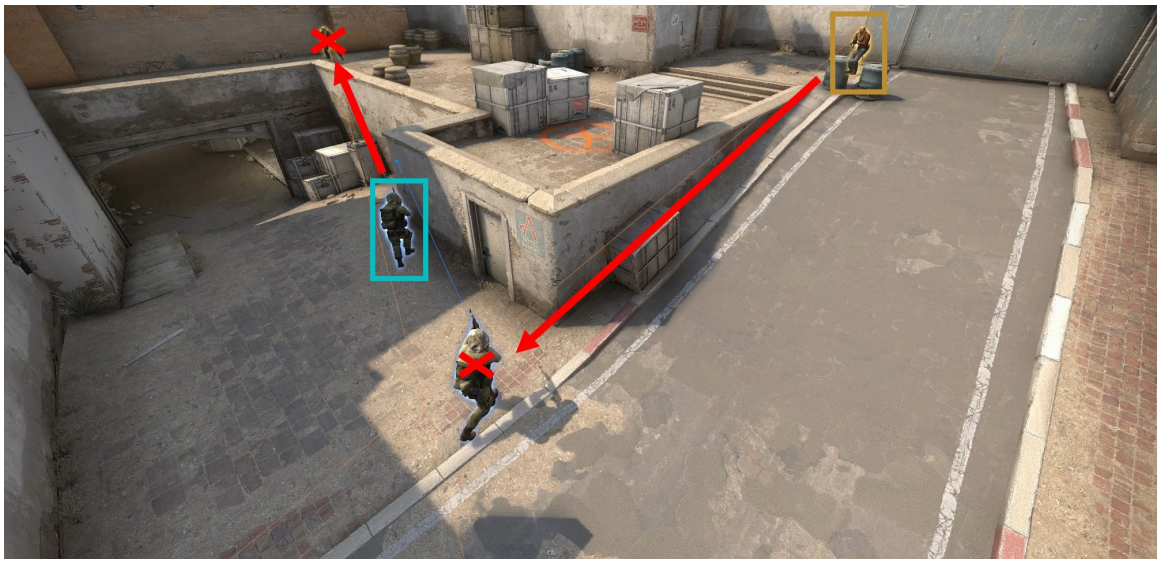


Figure 5.3: The game state can change rapidly in Counter-Strike. In this example, a yellow offense player and a blue defense player may be eliminated in the next second.

on a single CPU core.

5.1.1 Short Prediction Time Horizon

One of the key insights in this dissertation is that you can generate human-like, team-based behavior by training a model to make short-term behavior predictions. Leveraging this insight was our first key model architecture decision.

Scene Transformer predicts motion trajectories for up to five seconds, a standard time horizon for pedestrian motion prediction [73]. In their application, the state of the world changes slowly so the predictions are likely to be valid over the long time horizon. Counter-Strike is a fast paced game where state changes rapidly. In Figure 5.3, multiple players labeled with $\times$ may die in the next second. These rapid game state changes invalidate model predictions. We autoregressively output predictions over a short time horizon (375 ms or the next three time steps) to ensure our output is always valid for the current game state.

The short time horizon predictions also make our model more efficient. Long time horizon predictions have causal dependencies between the earlier and later tokens. Models from the autonomous vehicle domain predict tokens sequentially using a transformer encoder-decoder in order to learn these dependencies [73]. We can predict all tokens in parallel using only a transformer encoder since we do not have these causal dependencies between tokens over a long time horizon. Also, the shorter time horizon means our model outputs fewer tokens, reducing computational complexity. Figure 5.4 demonstrates this reduction with the crossed out output tokens.

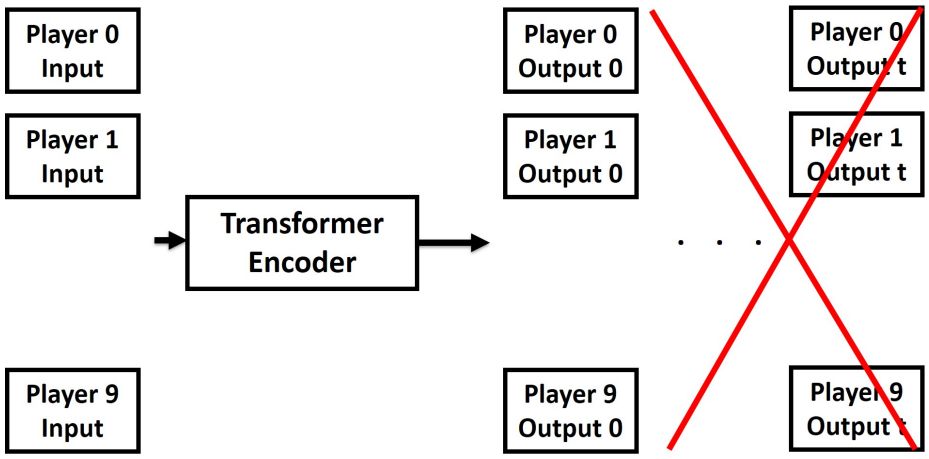


Figure 5.4: The shorter prediction time horizon reduces the number of output tokens. A transformer's computational complexity scales quadratically with the number of tokens, so fewer output tokens means our model is more efficient.

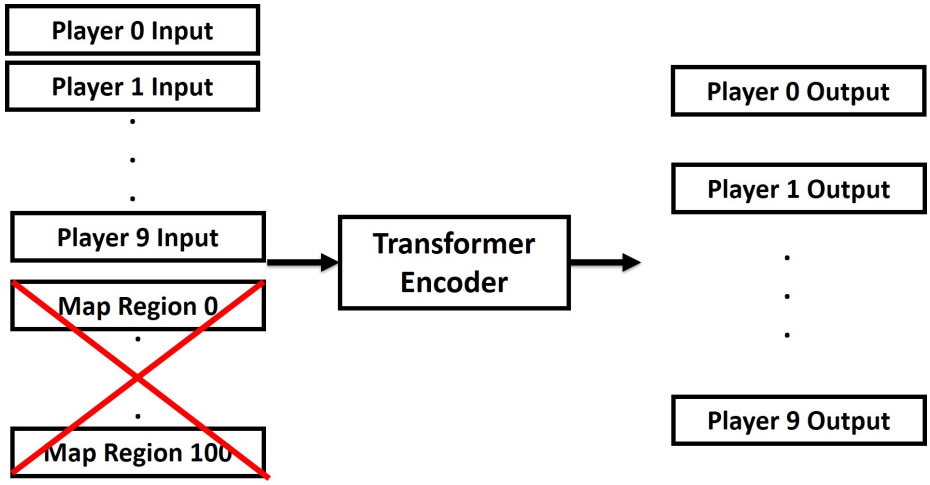


Figure 5.5: Specializing to one map reduces the number of input tokens. Reducing the number of input tokens reduces our model's computational complexity.

5.1.2 Specialize to One Map

The second key decision was specializing our movement model to one map. Multi-agent motion prediction systems for autonomous vehicles must generalize across different road geometries. Road conditions can change as vehicles travel from one part of the real world to another, and models must adapt to the changes. While Counter-Strike professionals play multiple maps, they tend to play and compete on the same game map for *years*, developing specialized movement strategies for each map's static geometry. So, it is perfectly reasonable to design movement models that are trained for

Figure 5.6: The action space modes can be sharp. This player must move in exactly the correct direction or they will fall off the ledge.

one map. This design choice allowed us to reduce the number of input tokens significantly, and as a result, the complexity of our attention layers, without impacting our model's applicability for our targeted use case. Note that each attention layer's complexity is proportional to the square of the number of input tokens [104]. To support multiple maps, we can pre-train our model for each map we want to support and make them available to our MLMove agent. (We would of course also have to curate a training set for each map as well, just like the training dataset for Scene Transformer includes data spanning multiple map regions.)

Figure 5.5 demonstrates the change to model input. Our model does not need the crossed out input tokens describing map geometry. Instead, it learns the map geometry by specializing to the map during training.

5.1.3 Sharp, Multimodal Action Space

The third key decision was designing the action space. Two properties of human movement in Counter-Strike impacted this design decision. First, players have multiple reasonable movement commands in many situations. The player in Figure 5.6 may walk forwards on their current ledge or jump up on the box to their left. However, the player is unlikely to walk to the right and fall off their current ledge. Our model's output action space must reflect this property by allowing a multimodal distribution of movement commands.

The second property is the sharpness of the modes. The player in Figure 5.6 is very close to the edge of the ledge, so the player is not likely to move directly forwards and risk falling off. Figure 5.7

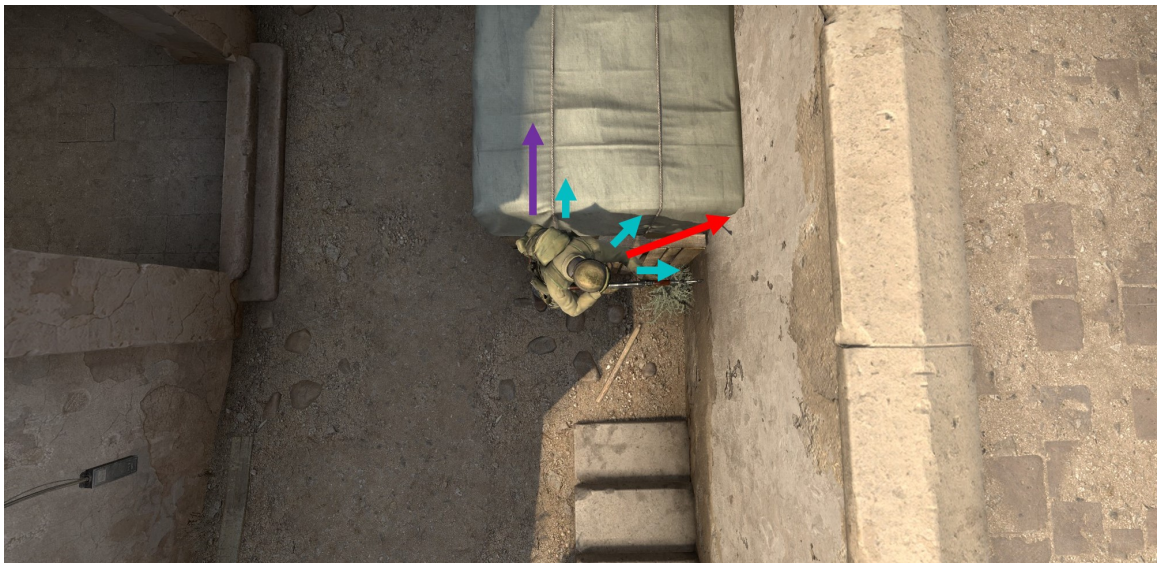


Figure 5.7: Our discrete action space allows for movement commands in multiple directions (blue arrows), multiple speeds (red arrow), and for multiple heights via jumping (purple arrow).

shows a top-down perspective of the player on the ledge. The player is not likely to walk in the directions of the blue arrows: falling off the ledge or walking into a wall. However, the player may walk slowly in the red direction to keep their balance on the ledge. Alternatively, the player may jump to the side to get onto the taller box, indicated by the purple arrow. Our model's output action space must allow picking the right combination of direction, speed, and verticality.

We use a cross entropy loss and discrete action space with 97 options to address these two properties of human movement in Counter-Strike. Section 5.2.2 describes the details of this action space.

5.1.4 Unmasked Attention Between All Living Players

The fourth key decision was allowing attention between tokens of all living players. Transformer's masking mechanism allows using hand-crafted rules to control attention between players' tokens. This control would have violated one of our key ideas: learning movement because human behavior is too complex for hand-crafted rules. Players interact with other players through a range of techniques including communicating with teammates, listening for audio cues, and studying opponents' habits. In Figure 5.8, the yellow offense player in the top right may be aware of the blue defense player connected by a red arrow, even though cover blocks sight lines between them, due to communication with the yellow offense player in the top left. We learn the interactions from the data. Due to our imitation learning approach, the attention mechanism learns the relationships between players that best predicts humans' next movement actions. The model will not use this information to gain an

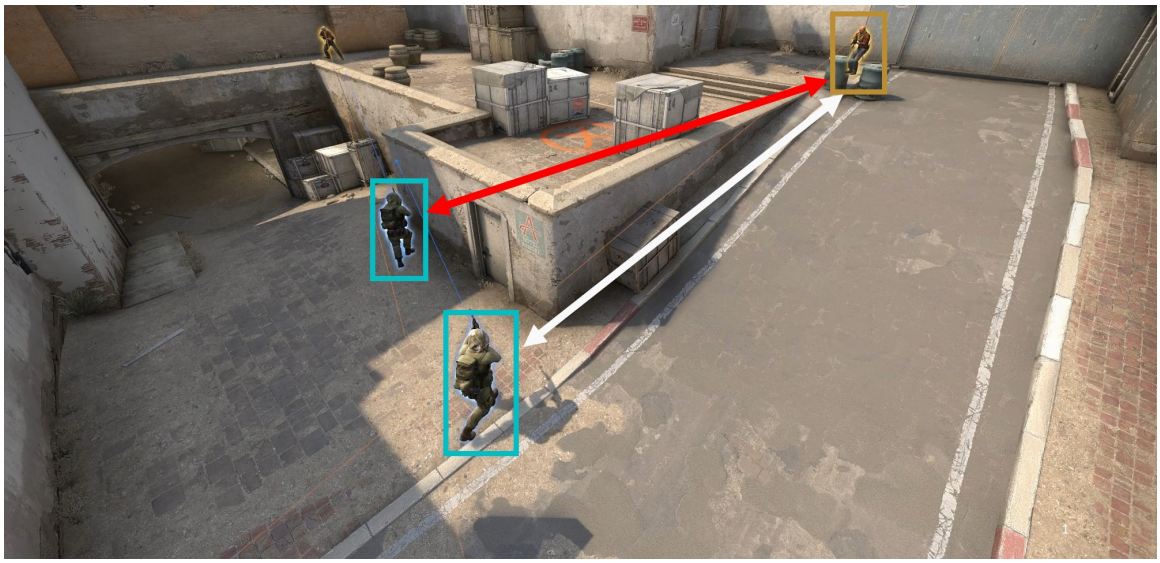


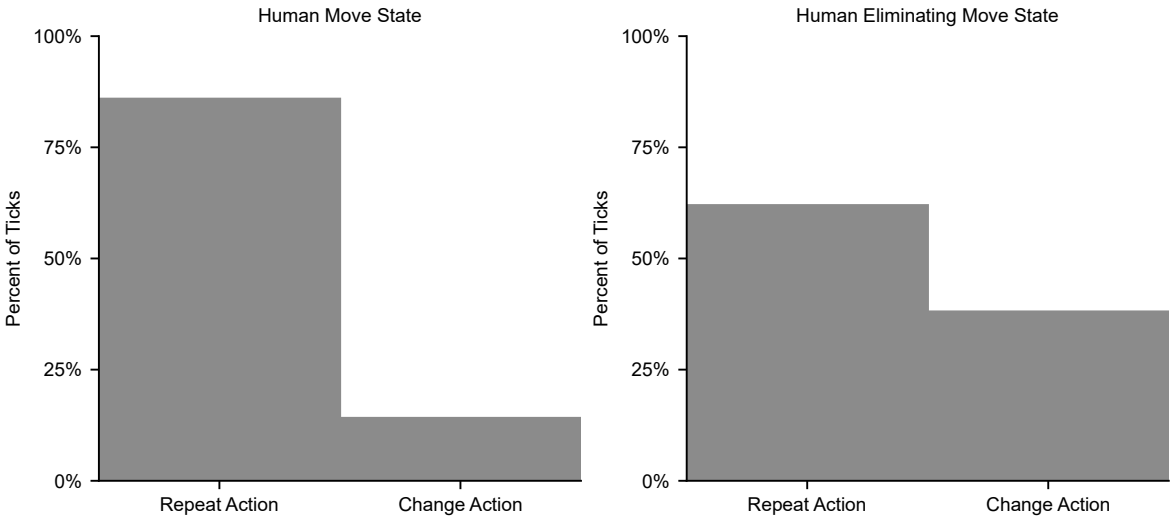
Figure 5.8: The yellow offense player in the top right may be aware of both blue enemies. The offense player can see the enemy connected with a white arrow. The offense player cannot see the enemy connected by a red arrow, but the other offense player may communicate this enemy's position.

inhuman advantage, since our objective is imitation rather than winning.

5.1.5 Only Input Current State

The final key decision was only providing the current input state to the model. A well known problem in imitation learning is the inertia problem, where models trained on sequences are biased to repeating recent actions, since this type of repeating "what I did last" behavior tends to dominate the dataset [14, 21, 94]. Figure 5.9a shows that across our entire dataset, players typically repeat their prior movement action.[1] This can lead to the failure to learn important movements like velocity change or (intentionally acted) "erratic" movements in combat, because they are both rare (low probability) events in the training dataset. Figure 5.9b shows that when players kill an enemy, they change movement actions much more frequently. We address the inertia problem using a simple solution that improves our model's prediction accuracy and efficiency: our model input consists only of "current" player states. The ablation in Section 7.3.5 shows our solution's effectiveness, as adding prior input states leads to less human-like map occupancy and kill location distributions. Additionally, only providing current input state improves model efficiency as the transformer's attention scales quadratically with the input length.

[1]To create the figure's binary analysis, we group all movement commands into one of two actions: moving or standing still.



(a) Across all game states, players typically repeat their prior movement action.

(b) When killing an enemy, players are more likely to change their movement action.

Figure 5.9: The frequency of repeating or changing movement actions.

5.2 Model Details

Figure 5.10 demonstrates the details of the learned movement model's transformer architecture. We explain those details in this section: input, output, and architecture.

5.2.1 Model Input

Our movement model's input is a sequence of 10 tokens, each token describing a player's current state. Counter-Strike logs contain up to 10 players at any time, up to five on each team. This is a broader range of players than in retakes. We train our model on 10 input tokens to enable it to generalize to a wider range of situations. The feature vector of each token is $[p_{i,t}, f_{i,t}, d_{i,t}]$, where $f_{i,t} = [l_{i,t}, u_i, b_t, x_t]$ and $d_{i,t}$ is a set of derived features that approximate information not contained in the game logs like visibility and team communication about strategy. Each token starts with the player's position $p_{i,t}$, alive status l_{i_t}, and team association u_i. We also include in each token the global map states of bomb location b_t and remaining time until bomb explosion x_t, information known to all players. We define the derived features in Section 4.2. We found that the derived features can aid attention in limited situations.

5.2.2 Model Output

Our movement model's output is a sequence of tokens, each token describing a player's movement command: which direction to move, how fast, and whether to jump. To capture the multi-modal

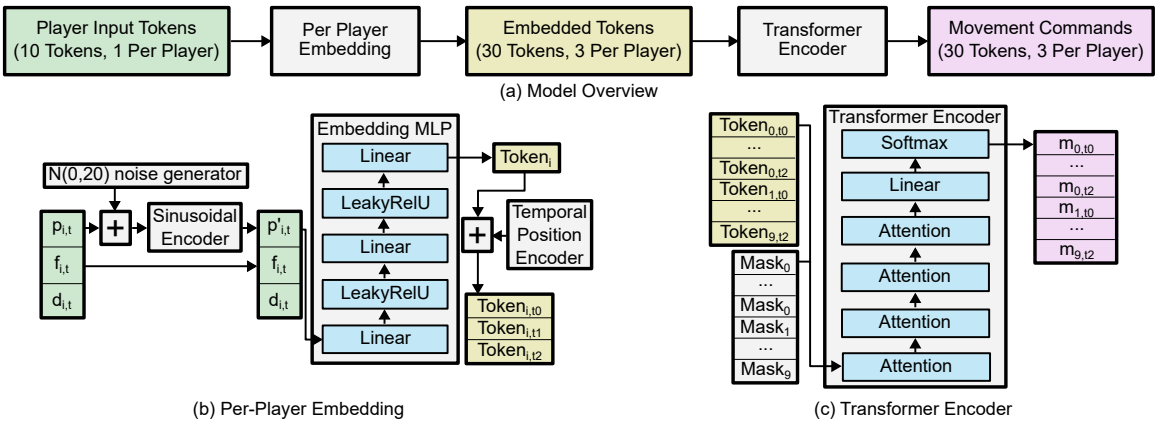


Figure 5.10: **The learned movement model.** (a) shows an overview of the two stages: (1) the per-player embedding stage converts the input tokens into embedded tokens, and (2) the transformer encoder uses the embedded tokens to predict the movement commands. (b) shows the per-player embedding stage that converts each player input token to three embedded tokens using a three layer MLP. (c) shows the transformer encoder that uses the embedded tokens and the associated masks to predict each player's movement command probabilities.

and stochastic nature of player movement, we represent a movement command as a discrete probability distribution with 97 options. Each option corresponds to a combination of one of 16 angular directions, three different movement speeds, two jumping vs not jumping states; plus a separate option for standing still. Movement commands are not recorded in Counter-Strike logs. We use heuristics to infer the movement commands from position/velocity information in the logs [79]. We found discretizing direction uniformly into 16 absolute angles is sufficient to navigate map details like thin ledges.

5.2.3 Model Architecture

The full architecture of the movement model is depicted in Figure 5.10(a). Each input player token is converted by an embedding network to an embedded token of dimension matching that of the transformer's attention layers; then each sequence of 10 embedded tokens corresponding to the states of 10 players are processed by the transformer to yield the movement command probabilities for the 10 players.

We use a learned embedding (Figure 5.10(b)) to convert input player tokens into vectors of dimension 256. Our embedding network consists of three linear layers, with LeakyReLU activations in between the linear layers. Our transformer encoder (Figure 5.10(c)), consists of four identical single-head self-attention layers of dimension 256. Like Vaswani et al. [104], we use a learned linear transformation and softmax to convert the outputs of our attention network to predicted probabilities of the output tokens (the player movement commands in our case).

To support eliminated players, we use transformer's masking feature. A transformer's attention layer computes the attention (connection) between all token pairs in the input sequence except for those that are masked out. So we set $mask(i,t) = 1$ for each token of an eliminated player ($l_{i,t}$ = false), to remove attention between that player and all other players. Also, we restrict the loss computation to only use $P(m_i)$ for players that are alive. Together with attention masking, this ensures eliminated players have no impact on our model's movement predictions for live players.

To learn temporally coherent motions, our model outputs predictions not only for the immediate next action (0 ms into the future) but also for actions at 125 ms and 250 ms from the current time. This is achieved by replicating each player's embedded token for time t three times, and summing each player's embedded token with the positional encoding of the three timestamps, to create distinctive embedded tokens for current time t, $t + 125$ ms, and $t + 250$ ms. Like Vaswani et al. and Ngiam et al. [73, 104], we use sinusoidal positional encoding for the player's in-game map position and for the three temporal positions represented as timestamps.

5.2.4 Model Training

We train the movement model using standard supervised learning (behavior cloning [81]) where we minimize the cross-entropy loss between the probability distributions of the predicted movement command and the ground truth movement command in the dataset.

We train using the CSKnow dataset (described Chapter 4) and perform an 80/20 train-test split: 5655508 train data points (98 hours at 16 Hz) and 1429953 test data points (25 hours at 16 Hz). Since there is a strong correlation between data points in the same round, we assign all data points in each round to the same subset. Once grouped into train/test subsets, we randomly sort data points irrespective of their round. We use the same train/test split for all training runs. To improve the model's ability to generalize, we add random Gaussian noise with mean 0 and variance 20 Counter-Strike units (less than a player's width of 32 units) to the player positions (see Figure 5.10(b)).

We train for 20 epochs with a batch size of 1024, an initial learning rate of 4e-5 controlled by the Adam optimizer with default configuration (βs = (0.9, 0.999), eps = 1e-08, and weight decay = 0). Training takes 1.5 hours on a single computer with a Intel i7-12700K CPU, 128 GB of RAM, and an NVIDIA RTX 4090.

5.3 Trade-Offs

We created an efficient, human-like movement model by combining a simple, imitation learning approach and a large dataset of human demonstrations. Our loss function only measures movement command accuracy over the next three time steps, not long-term consequences. One benefit of our behavior cloning approach is training efficiency. Our model trains in 1.5 hours on an RTX 4090 due

to the lack of interaction with an environment. A theoretical downside of our approach is that the model does not experience the long-term consequences of its actions during training. In Section 7.5 and Chapter 9, we will reflect on our approach's results and the potential value of more complex imitation learning methods [51, 72, 90, 105], which may improve long-term movement quality at the cost of slower training.

Chapter 6

MLMove Rule-Based Components

In Chapter 3, we explained that a human-like agent must move, aim, and shoot. We focused on movement during the last few chapters, but we cannot evaluate our movement controller without the rest of the agent. We also need to create human-like aiming and shooting controllers in order for our agent to play Counter-Strike with themselves and humans. In this chapter, we present the architecture of MLMove, our hybrid rule-based and learned agent. In Section 6.1, we provide an overview of the rules and how they integrate with the learned movement model in an industry-standard, hierarchical structure. In Section 6.2, we provide a detailed description of the rules and their hierarchical structure. In Section 6.3, we describe a derived feature that we compute as an input to the rule-based aiming controller.

6.1 Hybrid Hierarchical Structure

We use a modular approach shown in Figure 6.1 to integrate the learned movement model into MLMove. As explained in Chapter 1, hand-crafted rules can encode human-like aiming and firing behavior. Roughly two decades ago, games like Halo 2 and F.E.A.R. (described in Chapter 2) demonstrated how to organize these rules. We designed the hand-crafted part of MLMove using a hierarchical rule structure known as a behavior tree (BT), a technique popularized by Halo 2's developers [55]. The learned movement model's output is consumed by our BT.

The input to MLMove's BT is the current game state and the output is a sequence of game server commands. The server commands take the same form as commands sent by regular human players. Every 125 ms (16 game ticks), MLMove requests the learned movement model to predict the movement commands for all players using the current game player state as input. The agent caches and reuses the predicted movement commands for the subsequent 125 ms. The rule-based execution module is executed every game tick to emulate human mouse movement latency used for aiming. It converts movement commands at current time t made by our learned model into human

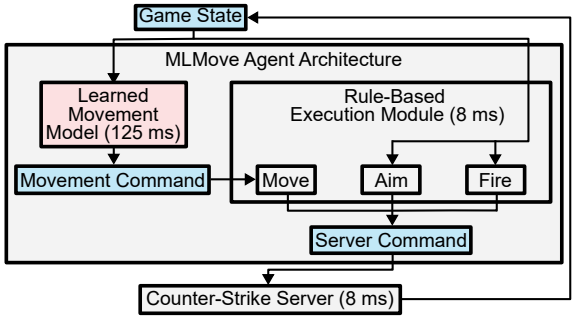


Figure 6.1: **MLMove Architecture:** MLMove uses the learned movement model to generate movement commands, then it uses a rule-based execution module to convert these commands into keyboard actions and also to generate aiming and firing commands. Counter-Strike server executes all player commands and sends the updated game state back to the agent.

players' keyboard navigation commands. The movement commands are only updated once every 125 ms to emulate human keyboard press latency.

The trained model predicts the movement for all players efficiently enough to be deployed in a commercial FPS game server. Specifically, the memory requirement for our model's 5.4M parameters is 21 MB; and the inference latency (time it takes predict the movement of all players) of our trained movement model deployed in C++ using LibTorch and TorchScript [83] is 8 ms with an IQR of 0.6 ms on a single core of an Intel Xeon 8375C CPU. Therefore, the amortized compute cost for our learned movement model for each game tick (frame) is 0.5 ms.

Our rule-based execution module also generates aiming and firing commands based on the current game state and player positions; the rule-based execution module sends all the "machine generated" game commands to the server, which will execute the commands and update the game state for the next frame.

For human and agent mixed play, the game server just replaces agent-generated commands with the corresponding human player's commands. As addressed in Chapter 7, we primarily test games consisting only of agents. However, the server allows any mixture of human and agent players for a maximum of ten players.

Aiming and firing We use standard techniques to handle aiming and firing. If no enemy is visible, the aiming module uses a probabilistic occupancy map to pick a target where enemies are likely to appear, emulating human-like predictive aim [56, 70]. If at least one enemy is visible, the module selects one target and tracks them until they are no longer visible. The aim module generates a smooth trajectory of view direction updates using a semi-implicit Euler method [3]. The fire execution module emits fire commands when the crosshair aligns with an enemy's axis-aligned bounding box. A distance-based lookup table controls the fire command frequency, shooting shorter and more controllable sequences at farther enemies that are harder to hit.

Figure 6.2: An example of the enemy probability distribution diffusion. The diffusion expands to fill non-visible regions. When players are visible to enemies, the distribution collapses to a single AABB.

6.2 Rule-Based Execution Modules

Our BT has three components: a team coordinator that handles long-term planning for where to aim, an individual player planner that translates the high-level plans into medium-term aim targets, and an action generator that converts the aim targets and learned movement model output into mouse and keyboard commands.

6.2.1 Team Coordinator

The highest-level component of the BT manages communication between members of each team. This component computes a team-wide probability distribution of enemy positions [56, 70]. Just like humans, agents know enemies' approximate positions at the start of a round and exact positions when the enemies are visible. Their position probability is assumed to diffuse uniformly through all non-visible parts of the map at the rate of maximum running speed.

Figure 6.2 demonstrates the enemy position probability distribution diffusion. There are three players in the image. The player in the middle is on defense. The players on the left and right are on offense. The colored lines show the probability distribution of each player's position according to the opposing team. Each line shows the diagonal of an AABB where that player could stand. (Counter-Strike API limitations restrict the number of lines we can draw, so we represent each AABB with a line.) The red lines correspond to the possible positions of the player on the left. They expand to fill the entire region on the left, because this region is not visible to the enemy in the middle of the image. The blue line corresponds to the possible position of the player in the middle. The green line corresponds to the possible position of the player on the right. The blue and green distributions

have collapsed to a single AABB each, since the opposing players can see each other.

6.2.2 Individual Player Planner

The mid-level component implements the team coordinator's decisions for each player. First, it selects the aiming target for each player. When enemies are visible, it selects which one to aim at. Players aim at an enemy's last visible location (known as a memory location) if no enemies are currently visible and either: (a) the remembered enemy was visible to the current player in the last 2.5 seconds or (b) the remembered enemy was visible to a teammate in the last second and in a position relevant to the current player. When no enemies are visible or remembered, it uses the enemy probability distribution to select an aiming target where enemies are most likely to appear. Section 6.2.2 provides more detail on how we use the probability distribution.

Danger Area Target Selection

The probability distribution defines enemies' likely positions. However, not all enemy positions are dangerous, like ones behind many walls. The probability distribution-based target selection process selects areas where enemies are likely to appear and hurt the current player. It uses a three step process to select an aim target.

First, it computes all the areas that are visible to the current players. Enemies can only hurt the current player if they are standing in these regions. Otherwise, the enemies will need to shoot through walls, which block bullets. We compute visibility using a standard technique known as a potentially visible set. The map is discretized using the navigational mesh's axis-aligned bounding boxes (AABB). Then, we trace a ray between the eye positions of a player standing in the center of each AABB. Two AABB are considered visible to each other if the rays can travel between the positions without hitting map geometry. Figure 6.3 demonstrates the potentially visible set for one origin AABB.

Second, it computes all the "danger areas," areas that are visible but next to non-visible ones. These areas are the most dangerous because they are the locations where enemies can appear. Figure 6.4 demonstrates the danger areas for one origin AABB.

Finally, it computes the most important danger area to aim at. Danger areas are sorted by multiple factors including: (1) how soon enemies could appear according to the enemy probability distribution, and (2) time since last aimed at by the current player or any teammate. This team-based temporal coordination ensures players check all danger areas as a team and minimize the odds of an enemy surprising them.

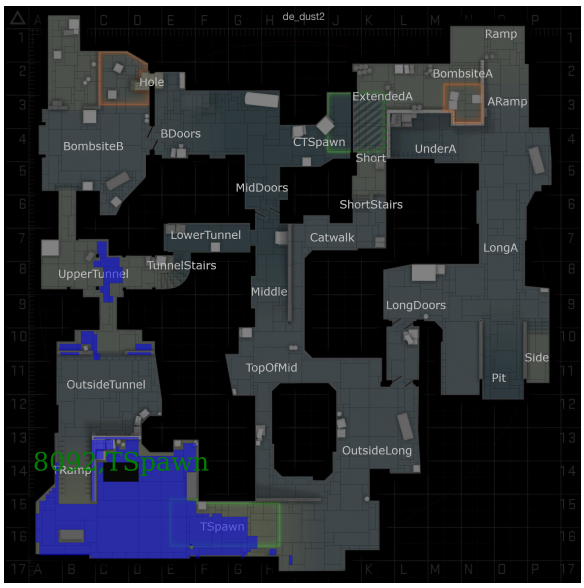


Figure 6.3: The potentially visible set of AABBs for a player standing in AABB 8092 in the bottom left of de_dust2. The blue AABB are likely visible. They gray ones are likely not visible.

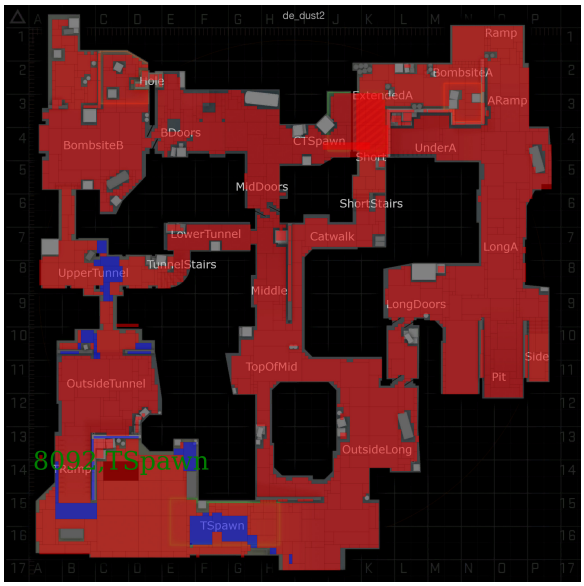


Figure 6.4: The danger areas for a player standing in AABB 8092 in the bottom left of de_dust2. The blue AABB are danger areas (visible AABB next to non-visible AABB). The red AABB are not danger areas.

Figure 6.5: The red boxes are the player positions tracked in a log file and in the network traffic. The bottom box is the bottom, center of the player's AABB. The upper box is the location of the player's camera. Each player's camera is behind their head. The player's camera is not blocked by the back of their head since the player's model is not rendered on their computer.

6.2.3 Individual Player Action Generator

The low-level component generates mouse and keyboard commands for each player. It aims at a target in a human-like manner with a semi-implicit Euler method [3]. If the player is aiming at a target, it generates distance-appropriate firing commands: shooting single shots at long distance and longer sprays during closer engagements. We account for recoil by exporting a recoil aim offset from the game engine. Finally, the action generator converts the A* path into keyboard commands.

This component also translates the learned movement model's movement command (produced once every 128 ms) into a keyboard movement action (produced once every 8 ms). The component uses the speed part of the movement command to determine if the keyboard movement action should include walking/crouching, specified by the shift/control keys. The component converts the absolute angular direction of the movement command to an angular direction relative to the player's current view angle, specified by the W/A/S/D keys. We find empirically that our absolute angular discretization made of 16 directions is sufficient to allow accurate in-game human trajectories, such as walking on thin ledges. The component uses the jumping part of the movement command to determine if the keyboard movement action should include jumping, specified by the space bar.

6.3 Derived Head Position

Like the derived features discussed in Chapter 4, we also compute useful features for the hand-crafted agent components. One of these features is head position. Skilled Counter-Strike players typically

aim at the head, since it drastically increases bullet damage. MLMove must know enemies' head positions in order to replicate this aiming behavior. Additionally, features like $cd_{i,t}$ (from Chapter 4) depend on head position.

Head position is not recorded in Counter-Strike's log files. The log files are recordings of the network traffic sent from servers to clients. This provides enough information to replay the game state in the engine. However, the logs do not store values exclusively stored in a server's or client's memory without being communicated to another computer. These values are reproduced by the engine during replay. The only way we can access non-serialized data during post-match analyses is by reverse engineering game logic.

Figure 6.5 shows the serialized player positions stored in a log file: players' camera positions above their shoulders and origins in the bottom-center of their axis-aligned bounding boxes (AABB). Head position is a function of these positions, a player's aiming direction, and their crouching state. The network traffic also contains values for computing head position using animation state, but it was not feasible to use these values. It was not technically feasible to reverse engineer the engine's entire player model animation code. Instead, we built an analytical model of head position.

6.3.1 Analytical Head Position Model

Our analytical model is an approximation of head position. The model has three steps.

The first step is finding an origin point. This point satisfies two conditions. First, it is a function of a value in the network traffic, so we can easily compute it. Second, it is a fixed distance from all head positions (regardless of where the player is looking). This property ensures we can compute a vector from the origin to the head position based on the player's view direction. Figure 6.6 shows that the point at the top of the torso and bottom of the neck satisfies both of these conditions. We compute this position using manually selected offsets from the player's camera position.

The second step is computing the offset vector from the origin point to the head position. The neck is a constant length and is (approximately) angled at the same position relative to the origin as the player's view angle. These factors mean that the origin-to-head vector is a constant length and just needs an angle derived from the player's view angle. Figure 6.7 shows that the model accurately computes head positions.

The third step is accounting for a player's crouching status. The player's crouching status changes the origin point and vector to head position. We address this issue by computing two sets of origins and offset vectors: one for standing and one for crouching. We linearly interpolate between these values using the player's crouch percentage to handle transitions between crouching and standing. Figure 6.8 shows the model accurately computes players' head positions when they are crouching.

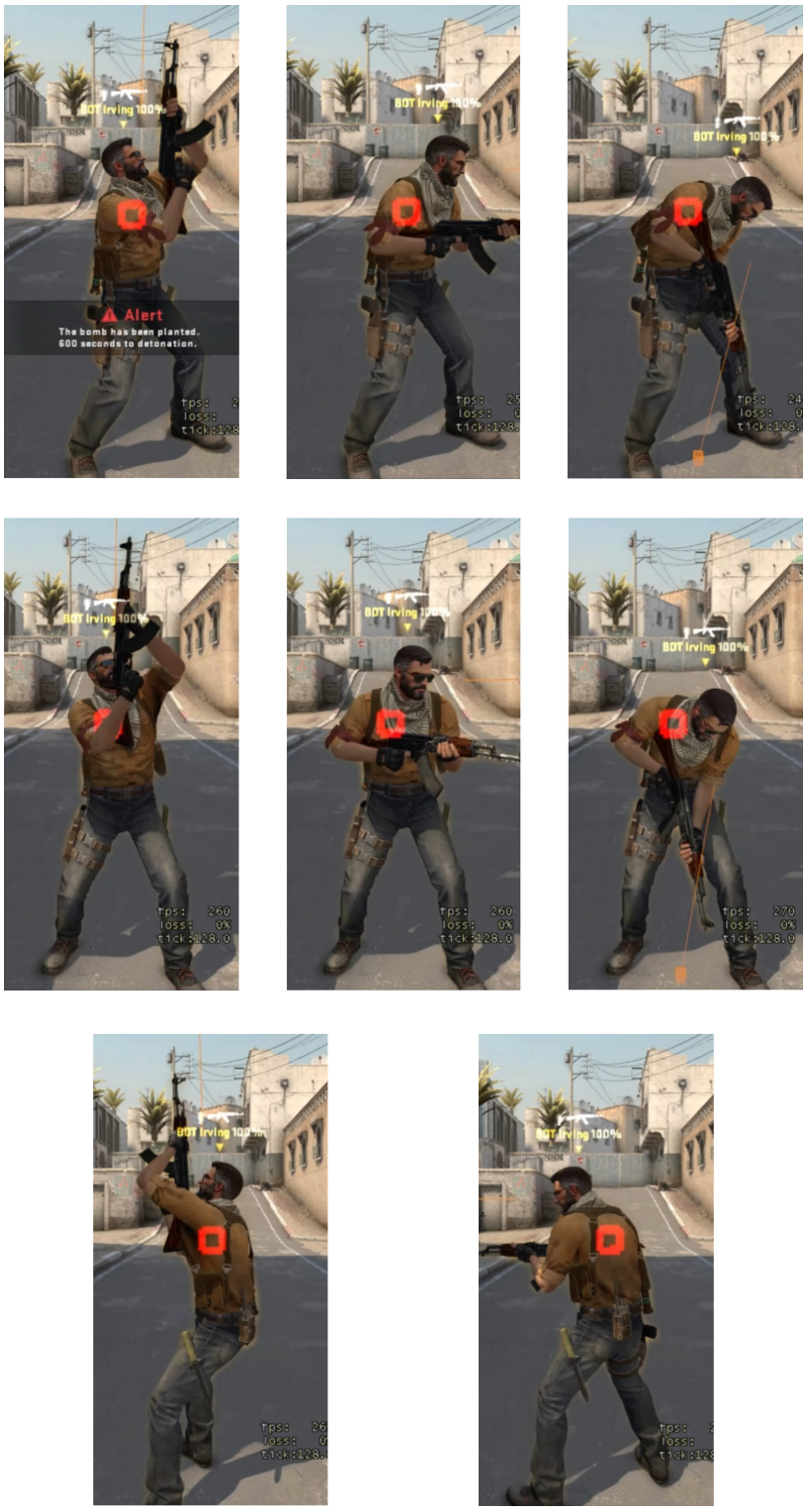


Figure 6.6: The point at the top of the torso and bottom of the neck (the red box) is a static offset from the player's camera and head positions regardless of where the player looks. Each row shows this is true for a different view yaw. Each column shows that this is true for a different view pitch.

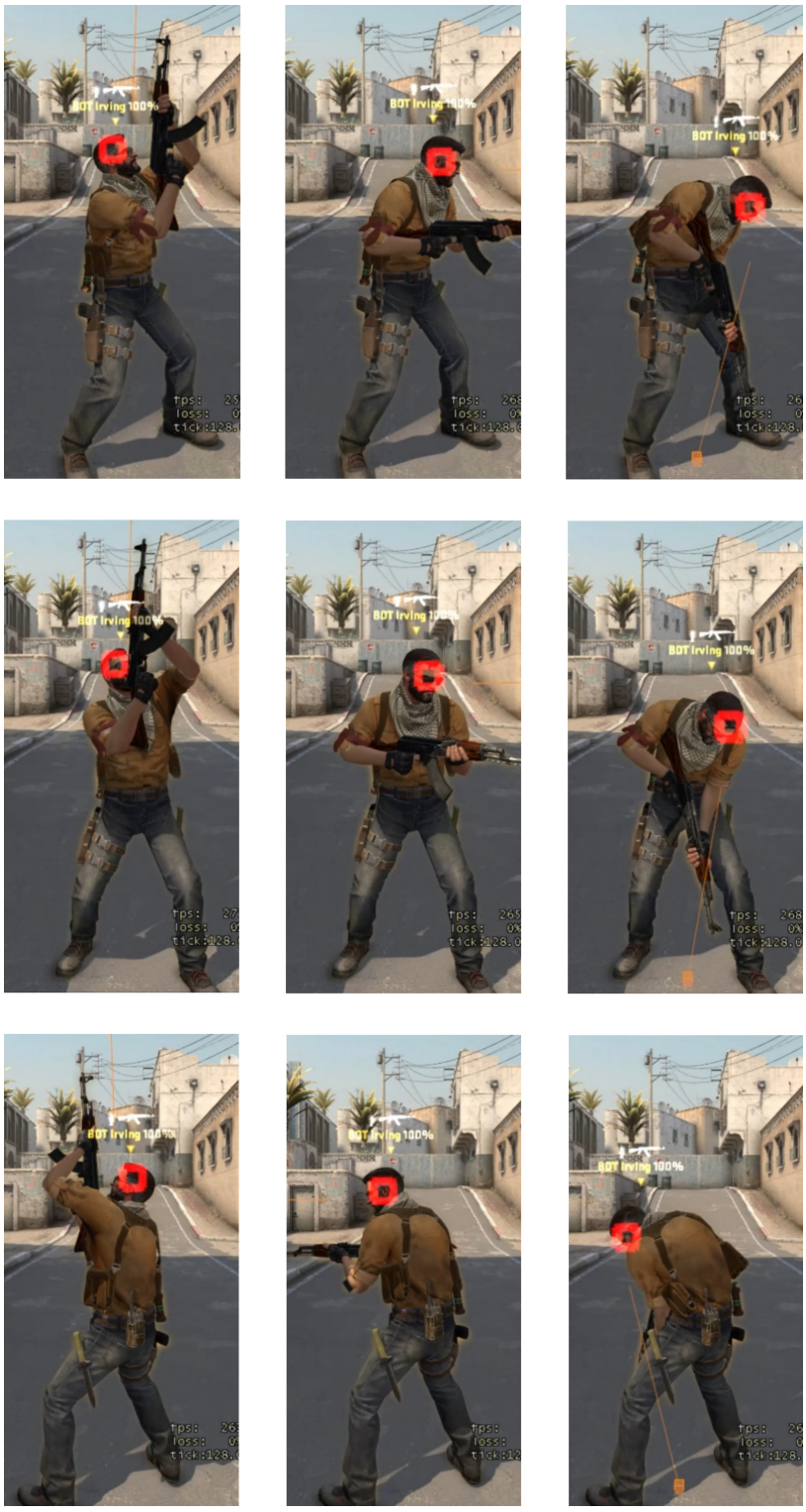


Figure 6.7: The red box shows the analytical model's computed head position. Each row shows the model is accurate for a different view yaw. Each column shows the model is accurate for a different view pitch.

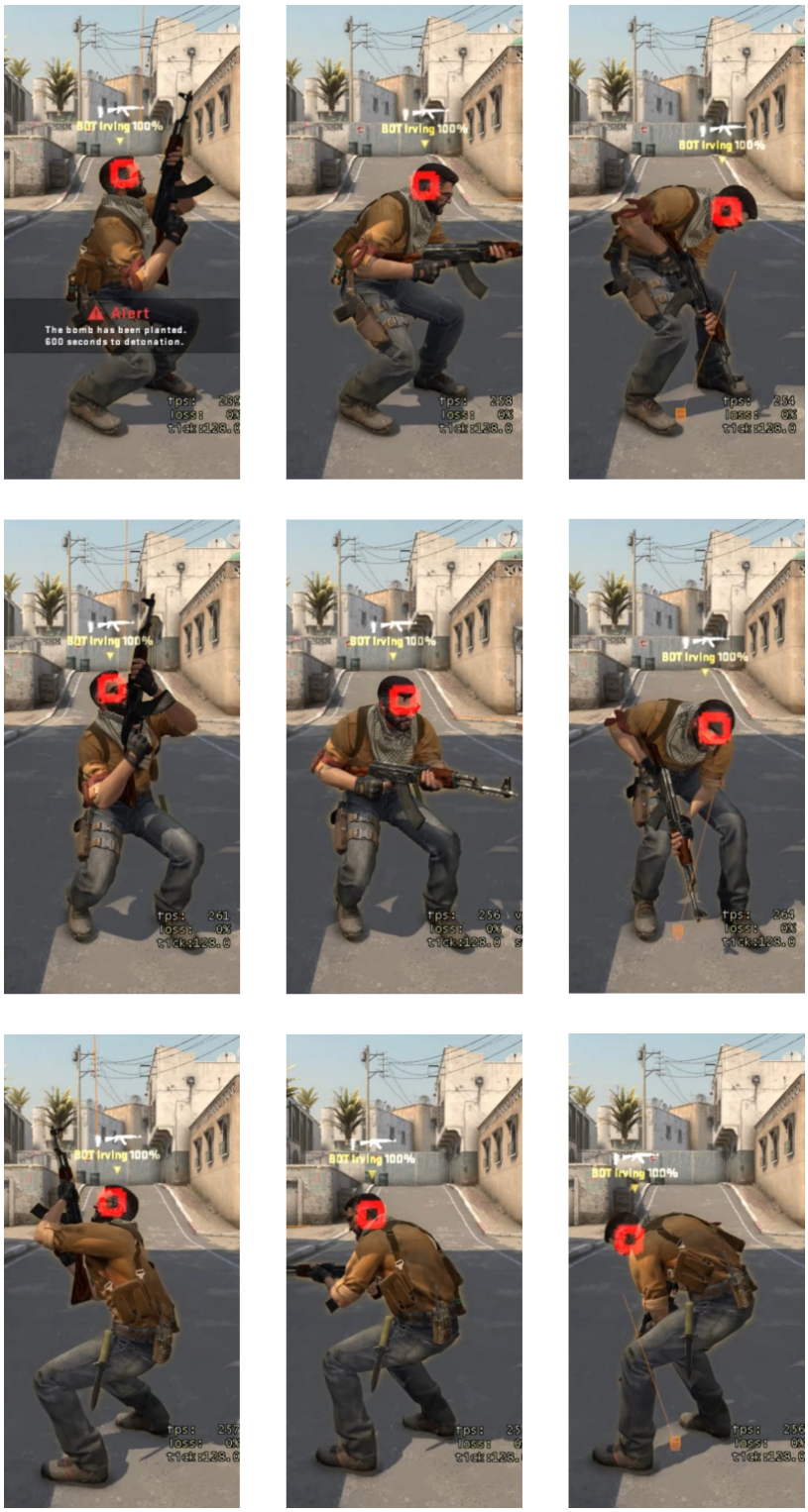


Figure 6.8: The red box shows the analytical model's computed head position when crouching. Each row shows the model is accurate for a different view yaw. Each column shows the model is accurate for a different view pitch.

Chapter 7

Evaluation

As we explained in Chapter 1, human-like behavior is very complicated. This complexity make evaluation difficult because there is no single metric of "humanness." Instead, we evaluate MLMove using a multifaceted approach. First, we conducted a small-scale user study (inspired by BotPrize 2010 [48, 49]) where human evaluators rank movement in videos of games played by humans and agents, and an exploratory study where humans play with and against the agents. Then, we performed a large-scale quantitative comparison on the distributions of movement trajectories and key outcomes from agent vs. agent self-play relative to those from professional human play. Through this combination of small-scale human ranking and large-scale quantitative analysis of outcome distributions, we present the first comprehensive evaluation of human-like team-based movement and other behavior for multiplayer FPS agents.

We start with the key results. Section 7.4 provides additional details on those results. Section 7.5 describes limitations of MLMove that we encountered through our evaluation.

7.1 Experiment Conditions

We compare four different player configurations:

- Human. Replay of the actual human data, taken from the CSKnow dataset.
- MLMove. Our agent with a learned movement controller, as described in Chapter 6.
- RuleMove. An agent with a rule-based movement controller implemented by the authors. The agent uses the same rule-based aim and firing controllers as MLMove. This agent was developed over several months by a skilled Counter-Strike player and should be considered a strong baseline for Counter-Strike agent design. Appendix A.1 provides further detail on this agent's logic.

- GameBot. The agent currently deployed in the commercial Counter-Strike game. Since it is third-party commercial software, the implementation details of this agent are unknown. It differs from MLMove and RuleMove in movement, aiming, and firing.

There are 1430 rounds in the CSKnow test dataset that meet retakes conditions. Our results analyze play from full retakes rounds where all players are controlled using the same player configuration, such as MLMove vs MLMove with no Human in the game. In the user study, we randomly sample 8 rounds across a range of initial conditions and record 32 videos, one for each combination of player configuration and round. For each round, participants viewed all four videos in a random order without labels identifying the player configuration. We used Counter-Strike to generate videos of game play, rendered from a birds-eye camera position and angle that best enabled analysis of team-based movement. For evaluator clarity, we used the "x-ray vision" rendering mode so evaluators can see players behind walls. The videos have a median length of 17 seconds and an IQR length of 17 seconds. We provide all 32 videos as well as the specific prompts of the study at `https://davidbdurst.com/mlmove/`. In the quantitative self-play experiments, we ran each player configuration through five iterations of all 1430 rounds in order to account for randomness in game play.

7.2 Human Assessment

To assess the realism of agent motion, we conducted a within-subjects study where we asked human evaluators to watch Counter-Strike game play videos depicting both human and agent play [78]. For each of the eight rounds described in Section 7.1, participants were asked to rank player configurations based on how well player movement matched their "expectation of how humans would move in that situation."

Evaluators. We recruited fifteen evaluators with Counter-Strike experience ranging from novice (never having played) to expert. Five of them achieved a rank of "Global Elite," the highest Counter-Strike player rating; and four had a rank of "Supreme Master First Class," the second highest.

Quantitative Ranking Results. Our study produces 120 rankings of the player configurations. Each ranking is an ordering of the four player configurations' similarities to expected human behavior in one initial condition according to one evaluator. To enable comparison between player configurations across all rankings, we use the TrueSkill rating [47] to aggregate the data into a single rating for each player. TrueSkill is a generalization of the Elo [37] rating system to multiplayer environments. In our work, a higher ranking means that a player configuration better matches the evaluators' expectations of human behavior.

In Figure 7.2 we plot the mean and standard deviation of the TrueSkill rating value for each player configuration. Unsurprisingly, Human achieves the best rating, whereas MLMove generates motion that matches evaluator expectations for human movement significantly more frequently than

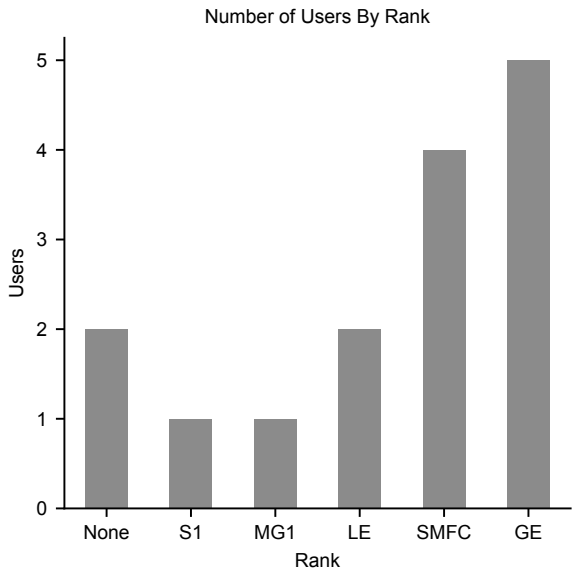


Figure 7.1: The user study contains participants with a diverse range of Counter-Strike experience.

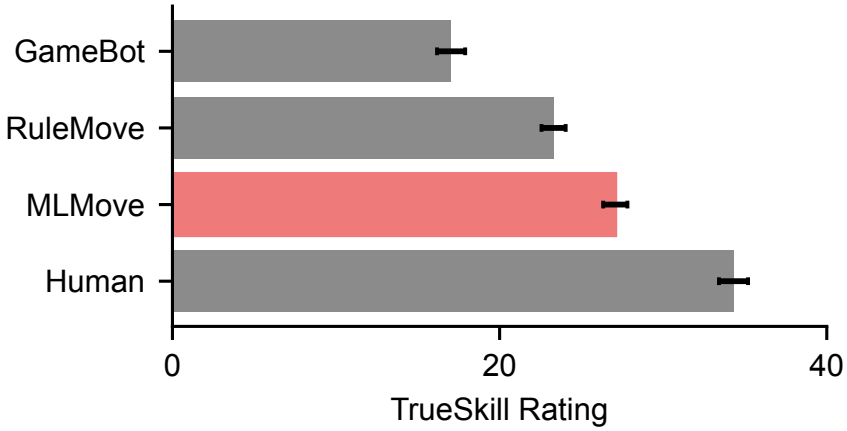


Figure 7.2: Human evaluators consistently rated MLMove's behavior as more human than RuleMove and GameBot.

the other agents. The results also suggest that RuleMove is a strong baseline, since it achieves a higher rating than GameBot, which is in commercial use today. The results are statistically significant according to a Kruskal-Wallis test (H=333, $p < 1e-5$) and Dunn post-hoc tests (all $p < 1e-5$).

7.2.1 Qualitative User Feedback

In addition to ranking the player configurations, subjects were also asked to explain their decisions. Expert subjects report that MLMove players demonstrated coarse-grained teamwork like "trading": killing an enemy while that enemy was distracted engaging someone else. Trading is a result of team-based movement, as two teammates must be in the right places at the right time to setup and take advantage of an enemy's momentary weakness. However, they also reported observing teamwork-related MLMove mistakes, such as being overly aggressive when trading, overly passive when supporting an attacking teammate, and lacking temporal coherence by rechecking previously cleared areas or jittering forwards and backwards. Experts complimented Human on their skilled collaborative movement, and criticized RuleMove and GameBot as agent-like. RuleMove was too rigid, and GameBot made illogical decisions.

7.2.2 Human vs. Agent Play

We performed an exploratory user study where experts play with and against the agents. Users clearly identified GameBot as the least human-like. Otherwise, the study was inconclusive because users said they were so engrossed in the game that they struggled to evaluate other players' movements in a short, highly controlled experiment.

7.3 Quantitative Self-Play Experiments Analysis

Beyond the user study, we provide a quantitative evaluation of the four player configurations by analyzing the statistics of full rounds of in-game self-play. Our metrics cover the key properties of movement: map coverage, utilizing expert strategies that avoid low-skill mistakes, and ensuring that movement yields key outcomes. The metrics covering mistakes and teamwork rely on a key insight: expert human language is the way to measure planning. We quantify teamwork and mistakes by utilizing expert labels for map regions, and then quantifying how humans navigate these regions in space and time. The use of expert terminology to formalize plans sets the stage for future work on foundation models to plan human movement using expert language. In Section 7.4.5, we quantitatively evaluate the ability of the learned movement model to reproduce humans' sequences of actions.

We perform our quantitative analysis on 1430 rounds (550 minutes) of MLMove game play; this is $\sim 5 - 16\times$ larger than the quantitative analysis on prior Counter-Strike agents by Pearce and Zhu [79] and on BotPrize agents by Gamez et al. [35], who inspired our use of Earth Mover's Distance and position-based metrics. For summary metrics, we report the median and IQR of the five round iterations discussed in Section 7.1. For distribution visualizations, we report results from the first iteration for each agent in order to compare distributions with the same numbers of rounds as Human.

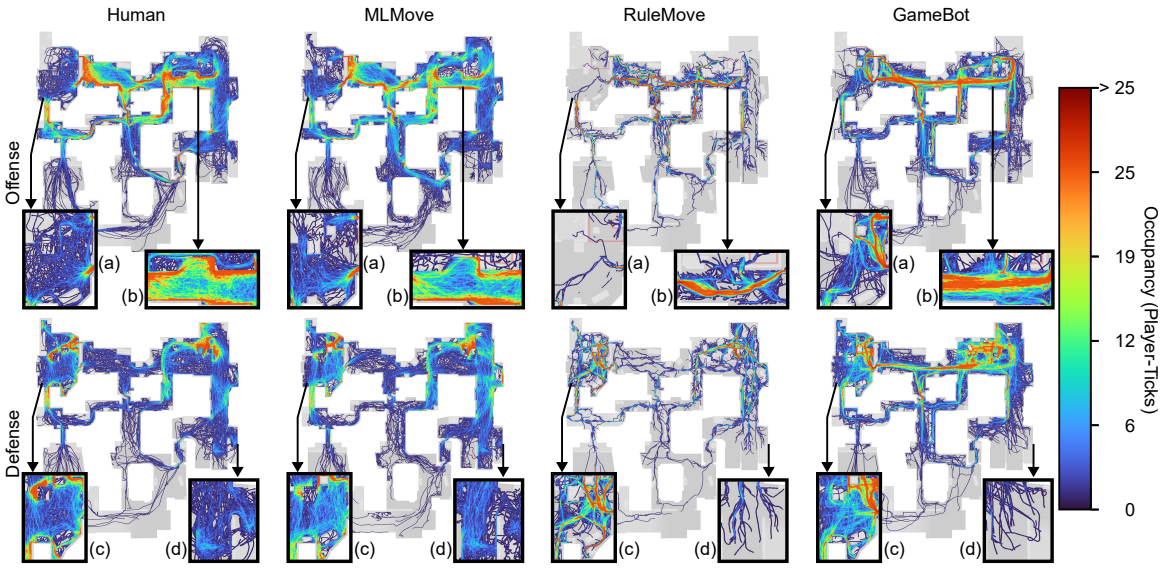


Figure 7.3: The fraction of time players spend in different regions of the map, aggregated over 1430 rounds of play. The distribution of the MLMove agents playing against themselves (second column) mimics the overall distribution of human play (Human, first column). A well-engineered rule-based agent (RuleMove) and the agents currently shipping in Counter-Strike (GameBot) do not replicate the human movement distribution.

7.3.1 Distribution of Player Positions

Figure 7.3 shows the distribution of player positions across the first iteration of 1430 rounds. Each pixel counts the game ticks when an offense or defense player occupies that location of the map. Overall, the distributions of the MLMove positions appear more similar to that of Human players than any of the other agent player configurations, for both the offense and defense teams. The first row in Table 7.1 shows that, when measured using earth mover's distance [61], the MLMove occupancy distribution (computed over both the offense and defense teams) is 1.8× and 1.9× more similar to that of Human play than RuleMove and GameBot respectively. We provide details of how we compute EMD in Section 7.4.3.

Figure 7.3 also shows that MLMove players exhibit skilled movement characteristics such as positioning themselves to remain out of enemy sight lines. For example, MLMoves stay near the walls on offense (inset (b)), and close to objects used for cover on defense (inset (c)), whereas the other agents traverse dangerous areas out in the open. MLMove players also demonstrate a greater diversity of behaviors than RuleMove, where each different behavior must be scripted. Insets (a) and (d) highlight examples where MLMove echos the diversity of real-world play, but RuleMove follows a limited set of predefined paths.

We also observe situations where MLMove produces movement that differs from the human trajectories in important ways. For example, inspection of Figure 7.3 suggests that the model fails

Table 7.1: Median ± IQR earth mover's distance (EMD) between map occupancy distributions (Section 7.3.1), player kill location distributions (Section 7.3.4), round lifetime distributions, and shots per kill distributions created from agent self-play and from real human data. In all metrics, self-play using MLMove yields distributions that are more similar to Human than RuleMove. We attribute the increased distance between lifetime distributions from MLMove and Human play to an increased number of long lifetime trajectories caused by instances of passive MLMove play (see Section 7.3.4).

EMD Type	MLMove	RuleMove	GameBot
Map Occupancy	**8.2 ± 0.5**	14.7 ± 1.7	15.2 ± 0.3
Kill Locations	**6.7 ± 0.1**	15.4 ± 0.7	16.4 ± 0.7
Lifetimes	4.9 ± 0.4	7.8 ± 0.0	**1.1 ± 0.0**
Shots Per Kill	**2.1 ± 0.1**	5.6 ± 0.0	4.9 ± 0.2

Table 7.2: Median ± IQR absolute percentage error (when counting instances of flanking and spreading configurations that arise out of teamwork) of agent players compared to human play data. MLMove more closely matches the human distribution of these multiplayer teamwork behaviors.

	Offense	Defense
MLMove	**27**% ±**22**%	**13**% ± **14**%
RuleMove	55% ± 29%	42% ±134%
GameBot	58% ± 23%	87% ±203%

to turn corners as sharply as Human. Human's inset (b) has more paths near bent walls than MLMove's because the humans can turn more sharply to follow the bends. We have found that MLMove's turning radius limitation is particularly detrimental in an area of the map that requires navigating consecutive tight turns followed by stairs.

7.3.2 Avoiding Common Mistakes

A first trait of "nonhuman" agent behavior is "a lack of common sense", which can be measured by the number of "common" mistakes. We consider two mistakes: (a) leaving high ground, and (b) giving up on an established defensive position. To characterize these mistakes, we identify specific combinations of players' positions within regions of the map indicating a defensive advantage on a game tick. For each such scenario, we compute whether the defensive players' regions in the next game tick indicate that they gave up their advantage. We measure the number of rounds with at least one mistake. As shown in Figure 7.4, MLMove's mistake rate is close to that of human players, and significantly smaller than those of the other agents.

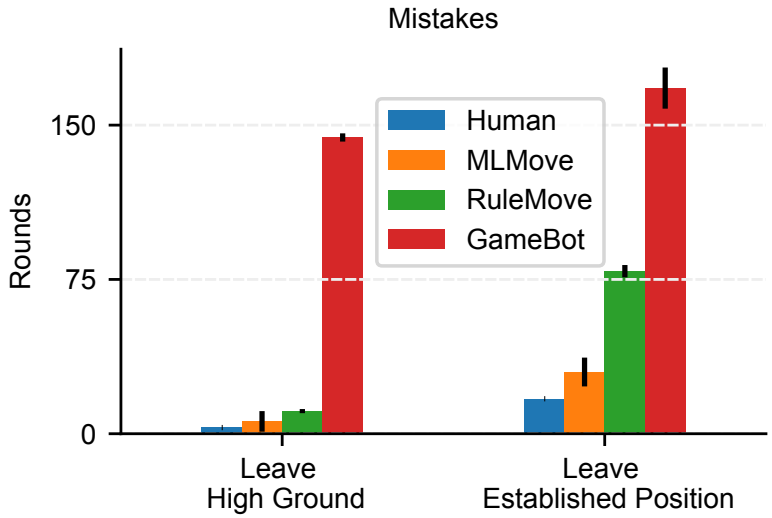


Figure 7.4: Median and IQR counts of rounds where at least one defensive player makes one of two common positioning mistakes (leaving high ground and leaving an established defensive position). MLMove makes these mistakes far less often than GameBot and RuleMove.

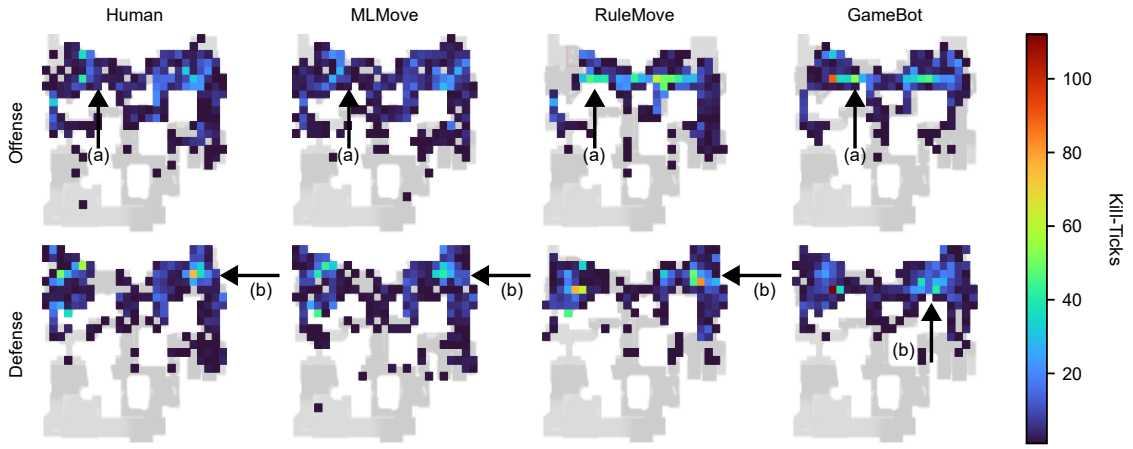


Figure 7.5: Visualization of the number of kills scored at each location on the map (position of shooter). (a) MLMove and humans avoid getting into combat in open areas, while RuleMove and GameBot frequently record kills from the center of the map, indicating bad positioning. (b) MLMove, RuleMove, and humans all score a high number of kills from positions of cover in the center of bombsite A, while kill locations of GameBot are more spread out.

7.3.3 Teamwork

We analyze the self-play rounds for instances of common forms of teamwork. Specifically we focus on *offense flanking*, where multiple players on offense approach the defense from different directions

to catch the defenders off guard. We also count instances of *defense spreading*, a tactic where defense players carefully distance themselves so that each player can cover a different potential attack direction, while being close enough to quickly reconverge on the most important actual attack direction.

We identify five unique two-player flanking configurations (involving different combinations of attack directions) and six unique three-player spreading configurations (covering different attack directions), and count the number of rounds where these configurations are observed. We define each configuration as a combination of the map regions occupied simultaneously by players on the same team. We compute the number of rounds with at least one occurrence of each configuration.

Table 7.2 shows that MLMove not only exhibits all five flanking and all six spreading strategies, but it also employs these strategies with a frequency more similar to human play than the non-learned agents. The median absolute percent error between human and MLMove flanking counts is 27%, far less than 55% and 58% for RuleMove and GameBot respectively. The median absolute percent error between human and MLMove spreading counts is 13%, far less than the 42% and 87% for RuleMove and GameBot respectively. See Section 7.4.2 for details on the definitions of and results for the individual flanking and spreading configurations.

7.3.4 Self-Play Outcomes

Skilled Counter-Strike players move to advantageous positions that increase the likelihood of eliminating enemies without being eliminated. We hypothesize that if MLMove moves similarly to human players, then we will observe similar distributions of where players are located when they score kills, how many shots are taken per enemy kill, and how long players live during rounds.

Kill Locations

Figure 7.5 plots the distribution of positions where players score kills (shooter locations), separated into offense and defense teams. Both humans and MLMove follow a cover principle when shooting enemies: they tend to shoot more frequently from positions that are protected. In Figure 7.5(a), both offense humans and MLMove avoid combat in the open areas leading to the B bombsite, whereas RuleMove and GameBot have poor positioning and engage in these cover-free regions. In Figure 7.5(b), defense humans, MLMove, and RuleMove (due to map-specific rules) primarily score kills from the center of the A bombsite, where the map contains objects that provide cover. On the other hand, GameBot scores kills uniformly around the entire bombsite. Row 2 of Table 7.1 quantitatively confirms that MLMove's kill location distributions are most similar to human play.

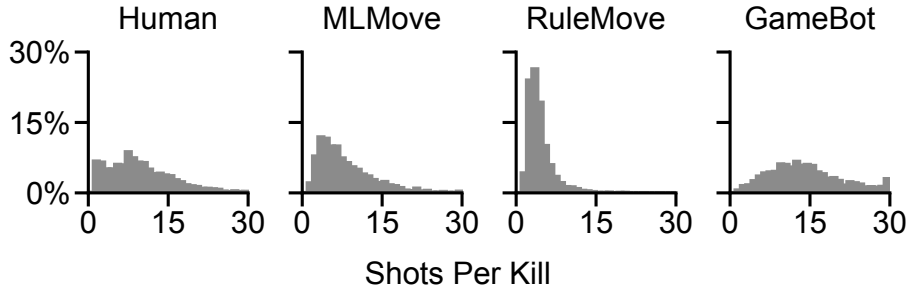


Figure 7.6: In Counter-Strike combat, players attempt to balance conflicting movement goals of staying still (to increase shot accuracy) and unpredictable movement (to avoid fire). MLMove reproduces the human distribution of shots per kill. RuleMove is scripted to stop prior to shooting, which leads to higher accuracy shots (fewer shots per kill), but contributes to shorter lifetimes.

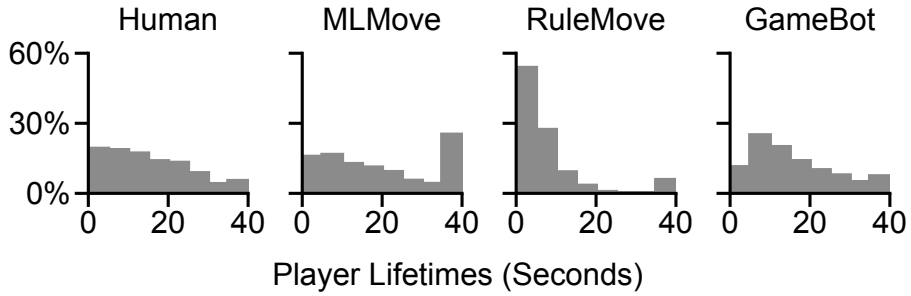


Figure 7.7: MLMove and GameBot reproduce the human lifetimes, while RuleMove's tendency to run at the enemy, regardless of the game state, leads to earlier deaths.

Shots per kill

We also observe that rounds involving MLMove-controlled players demonstrate approximately the same distribution of shots per kill as humans (Figure 7.6). Although it uses the same aiming and firing controller as MLMove, RuleMove produces a left-sided distribution, indicating fewer shots per kill. RuleMove's movement controller tells it to stop moving whenever an enemy becomes visible to increase its shot accuracy, but this behavior is not something all experienced human players would do in practice or in our training dataset.

Player lifetimes

Finally, we observe that rounds involving MLMove players exhibit a similarly shaped distribution of player lifetimes as that of human play (Figure 7.7). However, we also observe many more examples of MLMove players staying alive for the full 40-second period. We believe this is due to a conservative game play strategy present in the CSKnow dataset but not in the retakes test subset. A detailed analysis of this strategy is reported in Section 4.2 and Section 7.4.6.

Table 7.3: Median ± IQR EMD metrics for the ablated learned movement models. MLMove shows our movement model, NoAttn shows our movement model with all attention masked out, and History shows our movement model with prior state added to model input.

EMD Type	MLMove	NoAttn	History
Map Occupancy	**8.2 ± 0.5**	10.3	11.8
Kill Locations	**6.7 ± 0.1**	8.2	7.4
Lifetimes	4.9 ± 0.4	**4.6**	7.7
Shots Per Kill	2.1 ± 0.1	2.2	**1.2**

7.3.5 Ablations

We validate our movement model's design choices using ablations that compare the use of attention and the use of prior states versus without them. Table 7.3 shows results for our movement model (referenced as the default model in this section) in column 2, our model without attention (NoAttn) in column 3, and our model with prior player states added to the input (History) in column 4. Removing attention decreases model accuracy because the model fails to learn relationships between players which affect game play outcomes. NoAttn performs worst on Kill Locations, but also decreases model accuracy on map occupancy and shots per kill. Adding prior player states causes the inertia problem where players repeat their prior actions rather than responding to the dynamic changes in game states, resulting in worse map occupancy, kill locations, and lifetimes.

All models in Table 7.3 have a similar inference latency. Our default model has a median inference latency of 6.9 ms and IQR of 0.6 ms on one Intel 8375C CPU core, less than 8 ms. Ablations in Section 7.4.4 show that increasing the number of attention layers and the size of the MLPs inside each attention layer moderately improves map occupancy similarity to the Human distribution while increasing inference latency.

7.4 Additional Results

7.4.1 User Study

We can use the TrueSkill ratings computed in the study to predict the results of one-on-one comparisons. Figure 7.8 shows the probability of one player configuration being ranked as more similar to human behavior than another configuration. MLMove has an 11% probability of being ranked more similar to human behavior than Human.

7.4.2 Offense Flanking and Defense Spreading Details

Offense flanking is measured by recording instances when players occupy a configuration of positions simultaneously. For instance, when attacking BombsiteA, there are three attack vectors: from LongA, from CTSpawn, and from ShortStairs. Similarly, when attacking BombsiteB, the attack

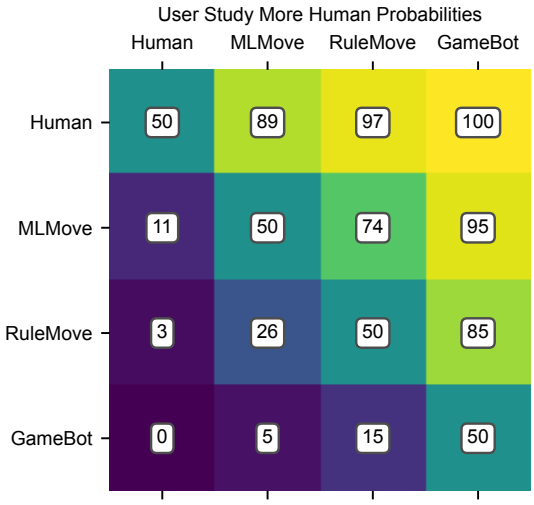


Figure 7.8: TrueSkill ratings predict MLMove will be rated as more human-like than Human 11% of the time and than RuleMove 74% of the time.

vectors are from BDoors, from Hole, and from UpperTunnels. We identify instances of flanking by noting the number of rounds with at least one tick where exactly two offensive players are alive and are positioned in two of the three different attack vectors for the target bombsite. This happens frequently in human data, occurring in 47% of the human rounds in the test dataset that start with two offense players alive. Figure 7.9 shows the error in the frequency of offense flanking configurations relative to humans. For almost all configurations, MLMove most closely matches Human frequency.

Defense spreading is measured in a similar manner. We have identified key locations where professional players spread out to counter flanks. When defending BombsiteA, the important defense positions are BombsiteA, LongA, and ExtendedA, which provide coverage of all offense attack vectors. Similarly, when defending BombsiteB, the key positions are BombsiteB, UpperTunnel, and BDoors. We identify instances of spreading by noting the number of rounds with at least one tick where exactly three defense players are alive and are positioned in key combinations of the important defense positions for the target bombsite. This happens frequently in human data, occurring in 45% of the human rounds in the test dataset that start with three defense players alive. Figure 7.9 shows the error in the frequency of defense spreading configurations relative to humans. For all configurations, MLMove most closely matches Human frequency.

Table 7.4 shows details of which map positions are in each configuration. The de_dust2 file defines the position labels (like LongA).

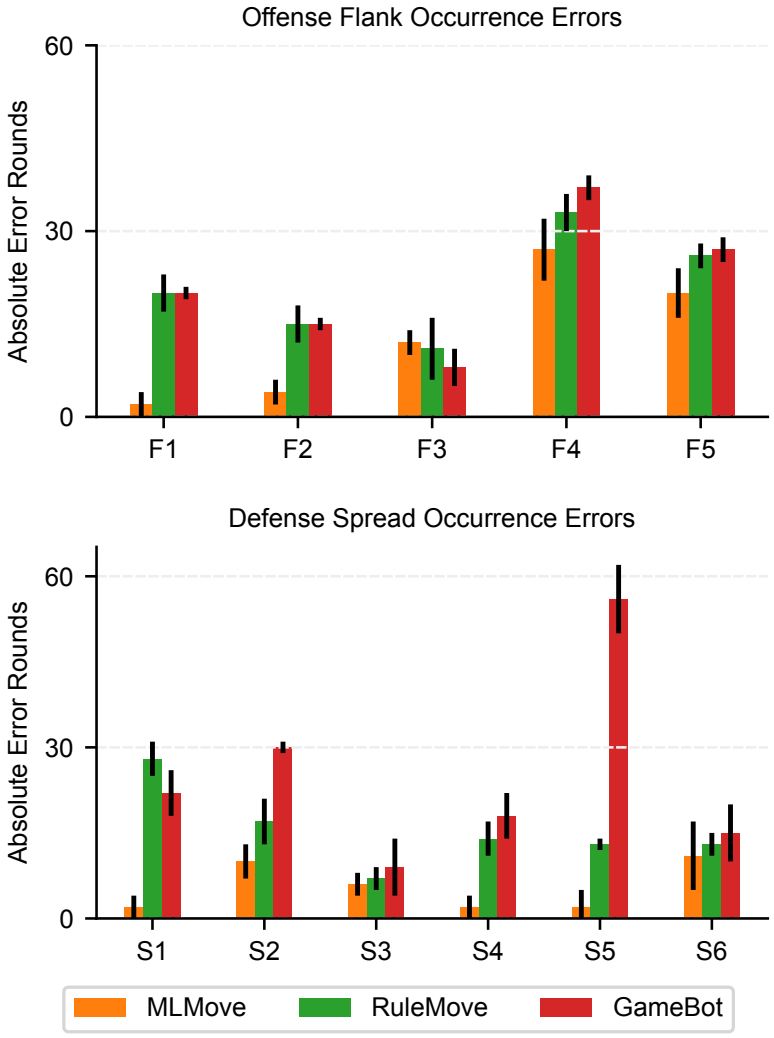


Figure 7.9: Median and IQR absolute errors of the number of rounds relative to HUMAN where agents on offense assume specified flanking configurations (F1-F5, top) and agents on defense enter specified spreading configurations (S1-S6, bottom). MLMOVE reproduces all of the examined flanking and spreading configurations, and does so with a more similar frequency to HUMAN than RULEMOVE and GAMEBOT.

Table 7.4: The teamwork configurations.

Configuration	Team	Bombsite	Player 1 Position	Player 2 Position	Player 3 Position
F1	Offense	A	ShortStairs	LongA	N/A
F2	Offense	A	ShortStairs	CTSpawn	N/A
F3	Offense	A	CTSpawn	LongA	N/A
F4	Offense	B	BDoors	UpperTunnel	N/A
F5	Offense	B	Hole	UpperTunnel	N/A
S1	Defense	A	BombsiteA	LongA	ExtendedA
S2	Defense	A	BombsiteA	BombsiteA	LongA
S3	Defense	A	BombsiteA	ARamp	LongA
S4	Defense	B	BombsiteB	BDoors	UpperTunnel
S5	Defense	B	BombsiteB	BombsiteB	BDoors
S6	Defense	B	BombsiteB	BombsiteB	UpperTunnel

Table 7.5: Model size ablation. Larger models moderately improve Map Occupancy performance while significantly increasing inference latency.

Metric	L4H1	L1H1	L1H4	L4H4	L16H1	L16H4	L4H1D256	L1H1D256
Map Occupancy EMD	8.2 ± 0.5	8.8	8.5	8.9	7.4	8.0	9.7	9.6
Kill Locations EMD	6.7 ± 0.1	7.3	6.1	6.3	6.3	6.8	6.2	5.8
Lifetimes EMD	4.9 ± 0.4	4.7	4.9	4.3	2.4	3.8	4.6	5.2
Shots Per Kill EMD	2.1 ± 0.1	2.1	2.4	2.5	1.5	1.7	2.1	2.3
Latency (ms)	6.9 ± 0.6	3.1	3.4	7.0	19.1	19.8	3.9	2.6
Model Parameters (M)	5.4	1.5	1.5	5.4	21	21	1.8	0.57
Parameter Size (MB)	21	5.7	5.7	21	81	81	6.7	2.2

7.4.3 EMD Calculation Details

We report the Earth Mover's Distance (EMD) between the distributions induced by the various agents through self-play and the ground truth human distribution. For EMD on two dimensional data, we downsample the map to a 120×120 grid (roughly 1 grid cell per player axis-aligned bounding box). The weights across these clusters are normalized to 1 to allow comparison between distributions with a varying number of points. We measure the EMD separately for offense and defense players, and report the combined values for both scenarios. For EMD on one dimensional data like lifetimes and shots per kill, we compute EMD directly on all of the data points.

7.4.4 Model Size Ablation

Table 7.5 shows that larger models run more slowly and moderately improve performance on Map Occupancy. We ablate the number of attention layers (l), the number of heads per layer (h), and the size of the MLPs after each attention layer (d). Transformer heads allow the model to learn multiple, independent representations by splitting the embedding space [104]. The default values in our paper are $l = 4$, $h = 1$, and $d = 2048$. Our results indicate that small models capture the coarse

details, and larger models capture more fine-grained aspects of movement.

7.4.5 Trajectory Matching in Isolation

We evaluate the ability of MLMove to generate human-like trajectories in the absence of confounding factors like aiming, firing, and kills. To accomplish this, we spawn players in a simplified, simulated environment where (unlike Counter-Strike) we do not take into account the presence of walls, players can instantaneously accelerate to maximum velocity or fall to the floor, and there is no aiming or combat. For agents initially spawned in the same positions, we compare the trajectories generated by the learned movement model and by other baseline methods with the ground truth trajectories stored in the CSKnow test set. These trajectories include save behavior and up to five players on a team.

We adopt standard metrics used in the automotive context [31, 73, 108]. These metrics quantify the capability of a probabilistic model to reproduce human trajectories over a short time horizon by picking the best rollouts over multiple trials from the same starting positions. More specifically we use minJADE (Minimum Joint Average Displacement Error, i.e., the average L2 distance between the corresponding points of the human and best-predicted trajectory over k trials) and minJFDE (Minimum Joint Final Displacement Error, i.e., the L2 distance between the last point in the human trajectory and the best-predicted trajectory of k trials). We modify the metrics to account for early player deaths in Counter-Strike by taking each player's final alive position in minJFDE and only including time steps when each player is alive in minJADE.

All the metrics are reported in Hammer units used in the Counter-Strike engine; as a reference, Counter-Strike players are 72 units in height [16] and their maximum speed is 250 units/second. We use $k = 6$ and split the trajectories into five seconds segments, matching with Ngiam et al. [73]. The resulting interval is long enough to allow players to enter a significant number of keystrokes, yet short enough to limit the effects of aiming, shooting, and killing on the trajectory. We perform auto-regressive rollouts for the five-second trajectories, using predictions at time step t as input at time step t+1. For all other input features, we use the ground truth values stored in the CSKnow test dataset throughout the entire rollout.

We compare trajectories generated by the MLMove learned movement model (second row in Table 7.6) with those generated by several baselines. These include one we identify as *Ground Truth Command*, where we measure minJADE and minJFDE while executing the movement commands stored in the CSKnow dataset in our simplified environment. This measures the error introduced by the assumptions adopted in our simplified environment; errors below this threshold should not be considered significant in this analysis. The other baselines are those typical of the related literature [10, 31] and are:

- *Stand Still.* The agent remains in its initial position. This policy represents players moving around inside a small region (e.g., in the case of Counter-Strike, a defender in the bombsite).

Table 7.6: Median ± IQR of MinJADE and minJFDE, measured in a simplified de_dust2 map while controlling the agents with several movement policies. The metrics of the *Ground Truth Command* policy indicate the error introduced by our simplifying assumptions.

Simulation Type	MinJADE	minJFDE
Ground Truth Command	42 ± 22	61 ± 36
MLMove (ours)	117 ± 78	186 ± 147
Stand Still	162 ± 120	276 ± 219
Starting Command	217 ± 138	437 ± 276
Nearest Neighbor	387 ± 265	477 ± 343

- *Starting Command.* Here, the agent continuously executes the first ground truth movement command stored in the five second sequence in the CSKnow dataset. This behavior represents players moving consistently towards an objective, or with a high degree of latency/autoregressiveness.
- *Nearest Neighbor.* We identify the most similar configuration of player positions in the CSKnow test dataset, and replicate the subsequent positions. This policy represents an agent controller that is purely based on a nearest-neighbor approach and lacks the capability of generating new trajectories.

Our simulator running the *Ground Truth Command* is sufficiently accurate to achieve a minJADE of 3.4% of the maximum distance that can be covered during a five-second rollout.

The metrics are reported in Table 7.6. We observe that MLMove outperforms the remaining baselines, suggesting that it generates trajectories with complexity that go beyond those of agents standing still or moving at constant speed and orientation. The metrics also highlight the nearest neighbor approach to be the worst for trajectory generation in this context.

7.4.6 Strategy Feature Accuracy

Sixty-seven of the 323 manually labeled push/save rounds are in the test dataset. During DTWADE-based nearest-neighbor push/save labeling, we only check for similarity with manually labeled rounds in the train dataset. As a result, we can use the 67 test rounds to evaluate the quality of our labels. We label each tick in these rounds using two sources: the manual labels and nearest-neighbor labels. Figure 7.10 shows the result of this analysis. Nearest neighbor is more accurate at labeling ticks that have a manual save label than a push one. Nonetheless, there are a significant amount of ticks with a nearest neighbor push label that are manually labeled as saves. This may bias the model towards saving behavior manifested as movement predictions that lead to longer agent lifetimes compared to human players.

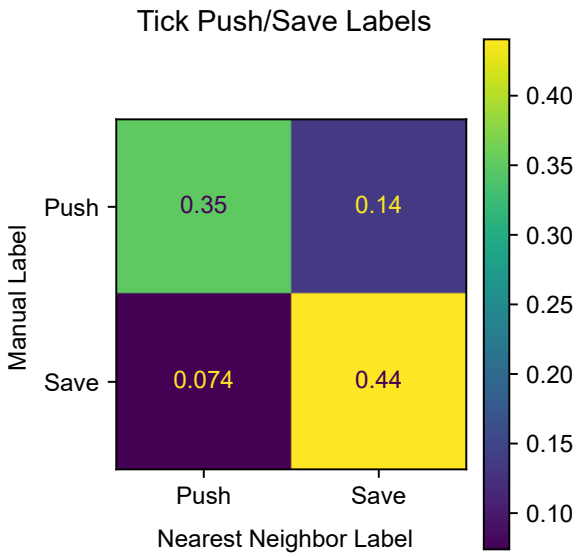


Figure 7.10: The nearest-neighbor push/save labels are reasonably accurate.

7.5 Limitations

Some of our model's limitations resulted from our design decisions, while others resulted from the scale limitations of academic research.

7.5.1 Design Decision Limitations

Two key design decisions may limit the quality of our model's behavior: only making short-term predictions and only inputting the current game state. We explained in Chapter 5 (and evaluated in Section 7.3.5) that inputting current game state addresses the inertia problem: always repeating the prior action because it is the dominant mode in the dataset. The inertia problem is a significant challenge for models trained to predict the next few actions, as human behavior is highly autoregressive over a short timescale. But, the lack of history means our model receives a qualitatively different input feature set than humans, and so generates some inhuman behavior.

The first inhuman behavior is microscale jitter. As noted by in Section 7.2.1, user study participants identified that MLMove sometimes rapidly moves back and forth while making progress towards the objective. This jitter is natural in some situations due to humans' multimodal movement distribution, but it is not normal when following a consistent trajectory over a long time period. Since the model does not receive history, it sometimes lacks the input features necessary to differentiate these two situations and produce a sharp movement command distribution.

The second inhuman behavior is saving. As noted by in Section 7.3.4, MLMove lives longer than humans. We have identified that this results from the saving behavior discussed in Chapter 4.

Saving is a strategy present in our data, but not in retakes mode, where players are willing to lose in order to survive. Saving is important for professional players in our dataset because they are playing a multi-round game mode rather than retakes, where dying in one round does not affect other rounds. Players will frequently save by hiding in a corner to avoid enemies. Hiding in the corner is one of the behaviors that exacerbates our inertia problem. The most likely behavior when in a corner is to keep standing still in a corner, even if the the global game state suggests that an another long-term strategy is required. Our short-term loss function does not explicitly prioritize these long-term strategies, and so the inertia problem leads to significant saving.

7.5.2 Scale Limitations

Our 123 hour dataset covers many of the game situations in Counter-Strike's retakes. It does not cover some fine-grained behavior present in human retakes gameplay. Additionally, we do not have data for other Counter-Strike game modes or other games.

Fine-Grained Behavior

Counter-Strike teamwork requires precise temporal coordination. If a player is a second too late in a flanking strategy, then their teammate may be eliminated before they are able to help apply pressure on the defense. As mentioned in Section 7.2.1, our user study participants noted that MLMove sometimes was too aggressive when executing team strategies. When we played with our agent, we also found temporal synchronization mistakes.

We believe that a larger dataset would enable our learned movement model to produce better temporal synchronization. The challenge is that human players, even professionals, are not always temporally synchronized. Looking through our dataset, offense human players did not always wait for their teammates before attacking. A larger dataset will enable the model to identify when temporal synchronization is human-like.

Other Game Types

Counter-Strike retakes always provides a clear objective: the bombsite. This clear long-term objective may simplify the learning problem. An open world game mode may require a larger model to determine the objective from ambiguous game state. Our short-term model receives the current objective (the bombsite) as an input.

Chapter 8

Agent API for Multiplayer FPS

Since Counter-Strike is not designed for agent research, we built the API necessary to train, test, and debug the MLMove agent described in Chapter 5 and Chapter 6. This API enables extracting game state from the engine, setting the game engine to particular states, and sending player commands to the engine. We implemented these features using bespoke game modifications. However, as agents become a greater part of games, we believe game engines should evolve additional APIs to make agent development more productive.

In this chapter, we describe the agent API we added to Counter-Strike. We hope future games use these features to influence their engine design. The obvious components of an agent API are reading game state data and issuing keyboard commands, which enable agent training and deployment. However, another critical aspect of agent development is testing. Agent testing requires that the API support creating isolated test scenarios and running headless at very high frame rates.

We recommend implementing the API using a game modification framework. We utilized a community driven Counter-Strike modification framework to implement our API [100]. By providing a game modification API, game developers can enable researchers to extend the agent API as needed and players to create new content. Even Counter-Strike started as a game modification for another FPS game [46].

In the last part of this chapter, we discuss how to test agents by combining the agent API with the behavior tree (BT) hierarchical rule structure described in Chapter 6. Testing requires more than just repeating game state at a high frame rate. A testing framework must also provide the ability to control the agents' internal state and specify success conditions. Our agents have internal memory, and we must reset that memory just like we reset the game state to create consistent tests. Our success conditions evaluate how multiple agents' positions evolve over time. The BT structure provides the temporal semantics necessary to specify these conditions. Since our agent's rule-based component is a BT, our BT testing logic can access the agent's internal state during testing. These properties make BTs and our agent API suitable for designing a testing framework.

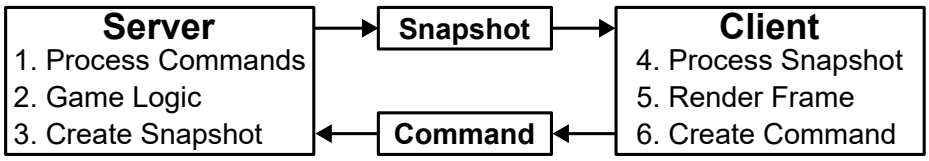


Figure 8.1: A simplified representation of Counter-Strike's distributed system architecture. The server updates the state and sends updates to the clients. The clients collect users' commands in response to the updates and send the commands to the server.

8.1 Agent API

The API must support three features: reading game state, setting the game to a specific state, and processing player commands. In order to train the agents, the API's read component must describe high frequency details of player behavior, contain key events like shots and kills, and enable the computation of advanced features like enemy visibility. To support agent testing, the API must enable repeatedly resetting game state to specific values so behavior can be evaluated under controlled conditions. To deploy agents, the API must enable processing the agent's commands.

There are two different sources of game state: game logs and a game server. The state reading component of the API is the same for game logs and a game server, so we can extract the same dataset from logs during training and from a game server during deployment. The state update components of the API are only implemented for a game server. Game logs are static objects, so updating them is not meaningful.

In some situations, analysts want to explore "what-if" scenarios where the logs are replayed with small modifications. The lack of state updates for logs may seem like a significant limitation for this application. For example, the Restrat Counter-Strike practice tool enables humans to update one players position while practicing against replays [87]. One could imagine updating multiple players' positions using human-like agents to provide a more responsive practice mode. Our API supports this application without updating state in game logs. We replay the logs and apply the state modifications in the game server.

8.1.1 Reading Game State

The API returns the game state data described in Chapter 4. We used the API to create our training dataset and extract the input features during deployment. Our API can read the game state data at 128 Hz, although we downsample to 16 Hz for the training dataset. The high frequency enables training our movement model to replicate human's fine-grained behaviors.

Our API does not include system state due to Counter-Strike's technical limitations. In this section, we will explain the ideal system state to log for games with a similar architecture to Counter-Strike. As shown in Figure 8.1, multiplayer FPS are frequently distributed systems with a single

authoritative leader, the game server, and many clients [24]. The server processes users' commands, updates game state, and sends the updates (known as snapshots) to the clients. Each player's client processes the snapshot, renders the new frame, records the user's command, and sends the command to the server. A round trip can require 100 ms. In order to make the system seem instantaneous, each client sees a different version of the world that depends on their network latency [25].

We recommend that future APIs return all perspectives, server's CPU performance, and clients' GPU performance. Counter-Strike current replay files only record the system's perspective or one client's perspective and omit processor performance. While logging this data is difficult, the data is crucial to analyze fine-grained behavior like reaction times and debug desynchronization or performance issues. One example use case of the data is training agents specialized to imitate humans playing on a poor internet connection. These agents would enable rapid testing of multiple lag compensation techniques. Poor logging infrastructure is one reason why developers struggle to develop and test lag compensation techniques.

8.1.2 Updating Game State

The API enables updating system state. There are two types of updates. The API allows directly updating many of the values specified in Chapter 4. We used this API component to implement behavior tests, like placing teams of agents in consistent positions and ensuring they always follow a reasonable path to an objective. The second update type is player commands. These commands replicate human mouse and keyboard actions. We used this API to deploy the agent, converting the movement model's outputs into actual in-game behavior.

A key API implementation challenge is handling dropped player commands. As shown in Figure 8.1, our API proposal separates the server and client into two separate processes. A client agent may fail to send commands on some frames due to spikes in model latency or other unforeseen errors. We handle this problem by replicating the prior command. This approach is acceptable given human perception. Our server performs a game step every 8 ms, roughly an order of magnitude faster than human reaction times [7]. If a command is dropped, the next one will be received well before a human can notice the drop.

8.2 Game Modification API

We implemented our API using a Counter-Strike game modification API. We recommend that future games support similar APIs to enable extensibility. There are three key requirements for a game modification API.

The first requirement is that the API can run user-defined logic every game step. This frequency is necessary to process player commands and log complete game state. We recognize that such an API brings performance concerns, as researchers may inject code that slows down every frame. APIs

may ignore user-defined logic that takes too long to run.

The second requirement is that the API support communication with external processes. This communication is necessary to interface with agents running in different processes, as shown in Figure 8.1.

The third requirement is that the API provide access to and documentation for internal game functions. We needed this functionality to implement key derived features and to debug our agents. We use the ray tracing internal functions to compute the enemy visibility features. We use the entity spawning feature to visualize and debug internal agent state, like the lines shown in Figure 6.2 that visualize players' probabilistic occupancy maps of enemy positions. Finally, we disable round end checking to create a debug mode. Typically, Counter-Strike retakes rounds end in 40 seconds, when the bomb explodes. Upon restart, players move to new positions. Our non-restarting debug mode allows testing without interruption.

8.3 Testing

An agent testing framework must support three capabilities: (1) reset game state and agent state to consistent values, (2) assert test conditions are true, (3) respond to human-controlled game state changes during human-agent mixed testing.

BTs and our agent API enable implementing these test components. Since all the test components are implemented as a BT, they can directly access and reset the internal state of the agent BT. They can set game state using the agent API. BTs provide temporal semantics for sequential and parallel composition of logic [55]. The sequential composition enables implementing temporal conditions, like following a leader at a reasonable distance. The parallel composition enables a player management subtree to continuously check for and respond to human-controlled game state changes.

We typically implement tests in this framework using four phases. First, an initialization phase enables testing isolated, individual behaviors, like following a leader. This phase freezes all agents, then sets their test-specific game state (i.e., positions) and internal state (i.e., aim targets). Next, a runtime phase allows the agents to play the game while verifying that constraints are satisfied. For leader-following tests, this phase may enforce a distance constraint between the players. Third, a validation phase ensures the test completes within a time constraint and that final conditions are met, such as the leader reaching a destination point. The fourth phase wraps the three other phases with a player management system that handles humans joining and leaving the server. If a human joins, the test may restart or player ids may be shuffled. This phase fixes the constraints so that the runtime and validation constraints apply to the correct agents.

Chapter 9

Conclusion

In this dissertation, we demonstrate an efficient, human-like agent for the multiplayer FPS Counter-Strike. Three key ideas drove the design of our agent. First, imitating human demonstrations at scale is the most direct approach to produce human-like behavior. In Chapter 4, we demonstrate our dataset curation system for extracting 123 hours of professional, human Counter-Strike demonstrations in a symbolic representation. This dataset is large enough to cover the entire space of possible game states in the Counter-Strike retakes mode. Second, autoregressive models trained to predict short-term actions can generate long-term, human-like behavior. In Chapter 5, we explain our efficient, transformer-based movement model trained using supervised learning to predict the next movement commands. This model generates human-like behavior, such as flanking and using cover, when deployed in our complete Counter-Strike agent. Third, we only use learning where necessary to create the complete agent. In Chapter 6, we describe MLMove, a Counter-Strike agent that combines the learned movement model with a behavior tree hierarchy of hand-crafted rules. We leverage domain knowledge to extract game state in an efficient, symbolic representation and design the behavior rules. Our evaluation in Chapter 7 shows our tiny transformer model is surprisingly effective when deployed in MLMove. We combine a user study, position metrics, and team strategy metrics to demonstrate that MLMove is more human-like than high quality, rule-based agent baselines. In the following sections, We will explain the implications of our results for multiplayer FPS agents, how future work may create even more human-like agents, and future applications.

9.1 Implications

Our results suggest that several commonly held beliefs about agents, particularly popular in the video game industry, are incorrect. Two of the most important are the value of symbolic game traces and the practicality of modern ML in compute-constrained environments like a multiplayer FPS. When we started, we believed that large ML models needed to process rendered game images

(or complex features derived from those images) in order to behave in a human-like manner. Our results suggest that massive datasets of symbolic game traces are sufficient to train small, efficient models of human-like behavior. We implore academic researchers and industry developers to create these datasets.

9.1.1 Efficient ML is Surprisingly Effective

Developers often assume they need to create massive models in order to process game state. In this dissertation, we have shown that modern ML techniques that generate human-like behavior are sufficiently efficient for modern games. Our movement model trains in 1.5 hours on a single GPU (attractive for modern game design workflows). The amortized inference cost per game step for our model is less than 0.5 ms on a single CPU core for all players, making it plausible for commercial game server deployment. The keys to our model's efficiency are (1) its GPT-style next token prediction and (2) the symbolic state representation. This approach works provides the model with complete game state (all players' positions) and uses a basic loss function (next actions) that forces the model to learn relationships between players.

While our work focused on Counter-Strike, the simplicity of the movement model architecture and training methodology should generalize to other multiplayer FPS games. We mainly leveraged common traits of FPS games in our design (rather than Counter-Strike specific features). However, for each new FPS game, a dataset of size and coverage similar to CSKnow is needed to train a transformer-based movement model similar to ours. There might be some game specific changes needed to the model input and output to match each game's navigation features. For example, another game might have four instead of three speeds or a few more complex movement modes (like climbing along ledges or ropes). We anticipate that small adjustments to the action space would enable a general, cross-game framework.

9.1.2 Symbolic Game Traces are Sufficient and Should be Recorded at Scale

Researchers have built small game play trace datasets [19, 110]. However, the game industry has the potential to create datasets that are multiple orders of magnitude larger. A single popular game, like Counter-Strike, can generate millions of hours of trace data every day [12]. If we had such a massive dataset, then we could easily generalize MLMove to new maps and games, as well as specialize it to individual player styles. But, developers need a good justification for the recording and storage costs incurred when creating such a large dataset.

Modern ML techniques can justify the cost of these large datasets. As explained above, we have shown that efficient ML techniques can generate human-like behavior. The idea that ML techniques are too computationally inefficient is a misconception. Efficient techniques can generate human-like

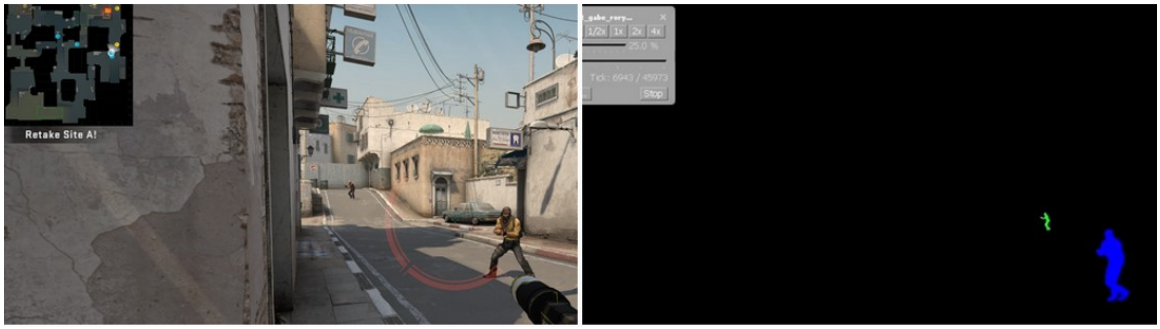


Figure 9.1: An early human behavior analysis experiment using complex feature engineering. The left image shows a normal Counter-Strike rendered frame. The right image shows the same frame where each enemy has a different color and everything else is black.

behavior without processing rendered images or complex features that approximate the images.

In the process of creating MLMove, we performed a series of experiments that emphasize the limitations of rendered images and complex feature engineering. First, we tried to identify cheaters by combining computer vision techniques with complex feature engineering heuristics (i.e. modifying Counter-Strike's rendered images to compute reaction time) [26]. Figure 9.1 shows an example of this feature engineering, where each enemy has a different color and everything else is black. This approach failed to identify cheaters since the game state representation did not account for the complexity of human behavior. It did not record why players might legitimately predict enemy positions based on communication with teammates or features of map geometry like cover.

A second experiment tried to improve the accuracy of the heuristics. We spent weeks reverse engineering Counter-Strike's network latency in order to compute reaction time features that are accurate to ~ 10 ms [25]. Again, more accurate heuristics did not address the fundamental problem: our hand-crafted features did not account for all possible game situations. It is not feasible to write behavior generation rules or analysis heuristics for every corner case [27]. Instead, modern ML approaches are necessary to learn feature embeddings that account for the full complexity of human behavior. Symbolic game traces are sufficient to train these approaches' models.

9.2 Principles For Future Agents

In addition to addressing our limitations, we envision that future agents will create more human-like experiences by building upon key principles of human behavior. These behavior principles are generalizations of the movement principles discussed in Chapter 3:

1. **Avoiding Exploitable Mistakes** - Humans can easily identify and defeat agents if they make basic mistakes like ignoring cover or repeating a behavior too often. Humans identified that RuleMove utilized the same flanking behavior too often. Even if a behavior is strong,

it can become weak if the enemy is expecting it.

2. **Responding to Enemies** - Multiplayer FPS are interactive experiences where enemies repeatedly counter each others' behaviors. The longer an agent can keep up with humans in this interactive chain of counters, the more human-like it will appear.

3. **Long-Term Planning with Game-Specific Expert Terminology** - For each multiplayer FPS game, humans create a custom set of terms for defining long-term strategies. These terms describe important positions in each map and how to coordinate behavior between players in different positions. We utilize Counter-Strike's map position and coordination terminology to evaluate agents' mistakes (Section 7.3.2) and teamwork (Section 7.3.3) like flanking.

We envision that future agents will generate even more diverse, difficult to exploit, and responsive behavior. New metrics may be required to evaluate exploitability. Agents may be deployed in games that last for hours, rather than 40 seconds like retakes mode, so players can explore the responsive behavior over extremely long timescales.

Future agents should also collaboratively plan strategies with humans using game-specific expert terminology. We anticipate that these long-term plans may address some of the limitations mentioned in Section 7.5. As explained in Chapter 4, we spent months trying to control long-term behavior like saving with hand-labeled features and heuristics. The expert terminology may serve as an improved long-term control signal. The terms can enable new input features and a long-term loss function to augment our short-term loss function. The new features and augmented loss may enable the agents to more accurately follow a long-term path without jitter and to decide when to leave saving corners.

A hierarchical modeling approach may automatically generate the terms to improve support for open world games. A large language model (LLM) could summarize the game state, predict the objective, and emit expert terminology to reach the objective. This hierarchical approach would combine the efficiency of a small model with the generality of an LLM. We look forward to playing with such agents.

9.3 Future Applications

It would be trivial to collect orders of magnitude more data in industry than our 123 hours. Human-like agents trained on these massive datasets may unlock a swath of new applications. Three of the most promising are personalized teammates, skill-appropriate practice partners, and inhuman behavior detection.

Players frequently play multiplayer FPS games with their friends. Human-like agents can serve as substitutes when friends are offline. Each player has their own style, so imitating a friend would require a significant dataset. Some players are too aggressive, while others are too timid. Each

player has their own favorite places to stand, and their own techniques for implementing the human-like behavior principles. Players know how their friends will respond to different situations, so they will be able to differentiate a general human-like agent from one tuned to their friend's style. A massive dataset seems necessary so human-like agents can train on demonstrations from the millions of players of a multiplayer game like Counter-Strike [12].

Training requires practice partners at the right skill level. Beginners need to experience the basic mistakes, how to avoid them, and how to take advantage of your opponent's. Human-like agents will never make these mistakes if they only play at a professional level. Additionally, players may get frustrated and stop training if they always lose to much better partners. If human-like agents can specialize to each skill level, then they can provide a range of skill levels from beginner to expert. A large dataset would provide demonstrations for training agents with these different skill levels.

Cheating players are difficult to detect due to the cat-and-mouse cycle of detection and generation. It is easier to generate human-like behavior than detect it. The cycle starts with anti-cheat developers spending a lot of time to identify cheating behavior, train a model to detect it, verify the model has few false positives, and deploy it. Cheating players are frequently skilled and know what looks suspicious, so they can easily change their behavior to invalidate the detectors and restart the cycle. Rather than creating cheating detectors, the ideal way to solve this problem is reversing the cycle. Anti-cheat developers have shown how to generate behavior that cheating players must detect, or the cheating players will self-identify and be easy to ban [28]. The key limitation with this technique was that it lacked a model of human-like behavior. We anticipate that human-like agents trained on massive datasets of human behavior will provide the necessary models and enable a new generation of cheat detection tools.

Appendix A

Appendix

A.1 RuleMove Rule-Based Execution Modules

We designed MLMove and RuleMove using a rule structure known as a behavior tree (BT) [55]. Our BT has three components, which we described in Section 6.2. In this appendix, we explain the rule-based movement logic in those components used only by RuleMove. We refer to Section 6.2 for the aiming and firing logic of the components shared between MLMove and RuleMove.

A.1.1 Team Coordinator

RuleMove uses the logic described in Section 6.2.1 for managing team communication about enemy positions. RuleMove also includes logic for managing long-term movement. The agent assigns each player a high-level path for attacking or defending an objective and an aggressiveness. For offense players, aggressiveness governs the order in which multiple agents travel the same path. For defense players, it governs how far the defenders position themselves from the objective, closer is safer but yields more territory to the offense.

A.1.2 Individual Player Planner

RuleMove uses the logic described in Section 6.2.2 for selecting an aim target. RuleMove also includes logic for non-learned movement. Based on teammates' positions, it selects the next waypoint on the high-level path. The module plans an A*-path to the waypoint using the map's navigational mesh. The path accounts for player aggressiveness using a spacing sub-component. The spacing sub-component ensures more passive players wait for more aggressive teammates to move first. Baiting players wait for teammates on the same long-term path. Lurking players wait for teammates on other long-term paths. Pushing players move first.

A.1.3 Individual Player Action Generator

RuleMove uses the logic described in Section 6.2.3 for generating mouse commands. The movement logic that generates keyboard commands is different. RuleMove picks the direction component of the keyboard commands given the current position and the target waypoint. It uses a quirk of Counter-Strike physics to jump. In Counter-Strike, players can accelerate while in the air. RuleMove jumps on obstacles by running into them, waiting until its velocity decreases to nearly zero, and then emitting a jump and crouch command. This heuristic ensures the agent has a controlled jump motion that reaches maximum vertical height and travels a small horizontal distance. We find that this heuristic is sufficient to navigate de_dust2.

Bibliography

[1] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 961–971, 2016.

[2] Görkay Aydemir, Adil Kaan Akan, and Fatma Güney. Adapt: Efficient multi-agent trajectory prediction with adaptation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8295–8305, 2023.

[3] Adam W Bargteil, Tamar Shinar, and Paul G Kry. An introduction to physics-based animation. In *SIGGRAPH Asia 2020 Courses*, pages 1–57. 2020.

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] The Viet Bui, Tien Mai, and Thanh H Nguyen. Imitating opponent to win: Adversarial policy imitation learning in two-player competitive games. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1285–1293, 2023.

[7] Robin Trulssen Bye. *The BUMP model of response planning: A neuroengineering account of speed-accuracy tradeoffs, velocity profiles, and physiological tremor in movement.* PhD thesis, UNSW Sydney, 2009.

[8] Dante Camarena, Nick Counter, Daniil Markelov, Pietro Gagliano, Don Nguyen, Rhys Becker, Fiona Firby, Zina Rahman, Richard Rosenbaum, Liam Atticus Clarke, et al. Little learning machines: Real-time deep reinforcement learning as a casual creativity game. In *EXAG@ AIIDE*, 2023.

[9] Ran Cao. Machine learning summit: Simulating teamfight tactics using deep learning for fast reinforcement learning ai training. Game Developers Conference, March 2023. URL `https://gdcvault.com/play/1029228/Machine-Learning-Summit-Simulating-Teamfight`.

[10] Ming-Fang Chang, John Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, et al. Argoverse: 3d tracking and forecasting with rich maps. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8748–8757, 2019.

[11] Panayiotis Charalambous, Julien Pettre, Vassilis Vassiliades, Yiorgos Chrysanthou, and Nuria Pelechano. Greil-crowds: Crowd simulation with deep reinforcement learning and examples. *ACM Transactions on Graphics (TOG)*, 42(4):1–15, 2023.

[12] Steam Charts. Counter-strike 2 - steam charts, January 2024. URL `https://steamcharts.com/app/730`.

[13] Filippos Christianos, Lukas Schäfer, and Stefano Albrecht. Shared experience actor-critic for multi-agent reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 10707–10717. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/7967cc8e3ab559e68cc944c44b1cf3e8-Paper.pdf`.

[14] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9329–9338, 2019.

[15] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[16] Valve Developer Community. Counter-strike: Global offensive/mapper's reference - valve developer community, September 2023. URL `https://developer.valvesoftware.com/wiki/Counter-Strike:_Global_Offensive/Mapper%27s_Reference`.

[17] Agentic.AI Corporation. Agentic — ai players as a service, 2023. URL `https://www.agentic.ai/`.

[18] NVIDIA Corporation. Nvidia t4 tensor core gpu for ai inference — nvidia data center, 2019. URL `https://www.nvidia.com/en-us/data-center/tesla-t4/`.

[19] FPS Critic. Pureskill.gg data science — pureskill.gg docs, 2024. URL `https://docs.pureskill.gg/datascience/`.

[20] Cecilia D'Anastasio. How influencers helped riot games turn valorant into a gen z hit, 2023. URL https://www.bloomberg.com/news/articles/2023-10-12/how-influencers-helped-riot-games-turn-valorant-into-a-gen-z-hit.

[21] Pim De Haan, Dinesh Jayaraman, and Sergey Levine. Causal confusion in imitation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[22] AAA Video Game Developer. private communication, January 2024.

[23] David Durst. Computing good crosshair placements using rlpbr's gpu ray tracing. 2021. URL https://davidbdurst.com/blog/csknow_cover_edge.html.

[24] David Durst. Analyzing online video games as distributed rendering systems. 2021. URL https://davidbdurst.com/blog/csknow_tracer_proposal.html.

[25] David Durst. Emulating csgo client-side demos with server-side demos. 2021. URL https://davidbdurst.com/blog/csknow_gotv_to_pov.html.

[26] David Durst. Challenges in computing reaction time from csgo demo files. 2021. URL https://davidbdurst.com/blog/csknow_demo.html.

[27] David Durst. Reaction times and crosshair positions aren't enough to catch cheaters. 2022. URL https://davidbdurst.com/blog/csknow_analytics_anticheat.html.

[28] David Durst and Carly Taylor. Online game technology summit: Hallucinations: Baiting cheaters into self-identifying by reversing detection. Game Developers Conference, March 2023. URL https://gdcvault.com/play/1029256/Online-Game-Technology-Summit-Hallucinations.

[29] Felix Endres, Jürgen Hess, Nikolas Engelhard, Jürgen Sturm, Daniel Cremers, and Wolfram Burgard. An evaluation of the rgb-d slam system. In *2012 IEEE international conference on robotics and automation*, pages 1691–1696. IEEE, 2012.

[30] Inc. Epic Games. Behavior trees in unreal engine — unreal engine 5.0 documentation, 2023. URL https://docs.unrealengine.com/5.0/en-US/behavior-trees-in-unreal-engine/.

[31] Scott Ettinger, Shuyang Cheng, Benjamin Caine, Chenxi Liu, Hang Zhao, Sabeek Pradhan, Yuning Chai, Ben Sapp, Charles R Qi, Yin Zhou, et al. Large scale interactive motion forecasting for autonomous driving: The waymo open motion dataset. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9710–9719, 2021.

[32] Meta Fundamental AI Research Diplomacy Team (FAIR)†, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray,

Hengyuan Hu, et al. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, 2022.

[33] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:18343–18362, 2022.

[34] Jane Friedhoff, Suma Bailis, and Feiyang Chen. Online game technology summit: Hallucinations: Baiting cheaters into self-identifying by reversing detection. Game Developers Conference, March 2023. URL `https://gdcvault.com/play/1034442/AI-Summit-Simulacra-and-Subterfuge`.

[35] David Gamez, Zafeirios Fountas, and Andreas K Fidjeland. A neurally controlled computer game avatar with humanlike behavior. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(1):1–14, 2012.

[36] Matt Gardner. Review: 'motogp 20' is a brutal experience, but one you'll want to try, April 2020. URL `https://www.forbes.com/sites/mattgardner1/2020/04/24/review-motogp-20-is-a-brutal-experience-but-one-you-want-to-try/?sh=5f53aead2533`.

[37] Mark E Glickman. A comprehensive guide to chess ratings. *American Chess Journal*, 3(1): 59–102, 1995.

[38] Ran Gong, Qiuyuan Huang, Xiaojian Ma, Yusuke Noda, Zane Durante, Zilong Zheng, Demetri Terzopoulos, Li Fei-Fei, Jianfeng Gao, and Hoi Vo. Mindagent: Emergent gaming interaction. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3154–3183, 2024.

[39] Dan Greenwalt. How forza's racing ai uses neural networks to evolve — war stories — ars technica - youtube, September 2020. URL `https://www.youtube.com/watch?v=XB9lf7iJbRw`.

[40] Xinying Guo, Mingyuan Zhang, Haozhe Xie, Chenyang Gu, and Ziwei Liu. Crowdmogen: Zero-shot text-driven collective motion generation. *arXiv preprint arXiv:2407.06188*, 2024.

[41] William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: a large-scale dataset of minecraft demonstrations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 2442–2448, 2019.

[42] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31, 2018.

[43] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[44] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[45] Jack Harmer, Linus Gisslén, Jorge del Val, Henrik Holst, Joakim Bergdahl, Tom Olsson, Kristoffer Sjöö, and Magnus Nordin. Imitation learning with concurrent actions in 3d games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[46] Joakim Henningson and Jack Ridsdale. From esports fever to cheating... counter-strike: Global offensive had it all, March 2024. URL `https://www.redbull.com/us-en/history-of-counterstrike`.

[47] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill™: a bayesian skill rating system. *Advances in neural information processing systems*, 19, 2006.

[48] Philip Hingston. A turing test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):169–186, 2009.

[49] Philip Hingston. A new design for a turing test for bots. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 345–350. IEEE, 2010.

[50] HLTV.org. Counter-strike news & coverage — hltv.org, January 2024. URL `https://www.hltv.org/`.

[51] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

[52] Daniel Holden, Oussama Kanoun, Maksym Perepichka, and Tiberiu Popa. Learned motion matching. *ACM Transactions on Graphics (TOG)*, 39(4):53–1, 2020.

[53] Chenguang Huang, Oier Mees, Andy Zeng, and Wolfram Burgard. Visual language maps for robot navigation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10608–10615. IEEE, 2023.

[54] W. Huang and D. Terzopoulos. Door and doorway etiquette for virtual humans. *IEEE Transactions on Visualization and Computer Graphics*, 2018. ISSN 1077-2626. doi: 10.1109/TVCG.2018.2874050.

[55] Damian Isla. Gdc 2005 proceeding: Handling complexity in the halo 2 ai, March 2005. URL `https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai`.

[56] Damián Isla. Third eye crime: Building a stealth game around occupancy maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, pages 206–206, 2013.

[57] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[58] Niels Justesen. Ai summit: Buffing bots with imitation learning. Game Developers Conference, 2022. URL `https://www.gdcvault.com/play/1027942/AI-Summit-Buffing-Bots-with`.

[59] Anssi Kanervisto, Joonas Pussinen, and Ville Hautamäki. Benchmarking end-to-end behavioural cloning on video games. In *2020 IEEE conference on games (CoG)*, pages 558–565. IEEE, 2020.

[60] Saman Kazemkhani, Aarav Pandya, Daphne Cornelisse, Brennan Shacklett, and Eugene Vinitsky. Gpudrive: Data-driven, multi-agent driving simulation at 1 million fps. *arXiv preprint arXiv:2408.01584*, 2024.

[61] Soheil Kolouri, Se Rim Park, Matthew Thorpe, Dejan Slepcev, and Gustavo K Rohde. Optimal mass transport: Signal processing and machine-learning applications. *IEEE signal processing magazine*, 34(4):43–59, 2017.

[62] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[63] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning*, pages 80–93. PMLR, 2023.

[64] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and brian ichter. Chain of code: Reasoning with a language model-augmented code emulator. In *Forty-first International Conference on Machine Learning*, 2024. URL `https://openreview.net/forum?id=vKtomqlSxm`.

[65] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[66] David Lind. Game server performance on 'tom clancy's the division 2'. Game Developers Conference, August 2020. URL `https://gdcvault.com/play/1026706/Game-Server-Performance-on-Tom`.

[67] Hung Yu Ling, Fabio Zinno, George Cheng, and Michiel Van De Panne. Character controllers using motion vaes. *ACM Transactions on Graphics (TOG)*, 39(4):40–1, 2020.

[68] Jinming Ma and Feng Wu. Learning to coordinate from offline datasets with uncoordinated behavior policies. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1258–1266, 2023.

[69] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *2nd Workshop on Language and Robot Learning: Language as Grounding*, 2023. URL `https://openreview.net/forum?id=WbFBZ3bQUs`.

[70] Hans P Moravec. Sensor fusion in certainty grids for mobile robots. *AI magazine*, 9(2):61–61, 1988.

[71] Yusuke Mori. Ai summit: Developing adventure game with free text input using nlp. Game Developers Conference, March 2023. URL `https://gdcvault.com/play/1028928/AI-Summit-Developing-Adventure-Game`.

[72] Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[73] Jiquan Ngiam, Vijay Vasudevan, Benjamin Caine, Zhengdong Zhang, Hao-Tien Lewis Chiang, Jeffrey Ling, Rebecca Roelofs, Alex Bewley, Chenxi Liu, Ashish Venugopal, et al. Scene transformer: A unified architecture for predicting future trajectories of multiple agents. In *International Conference on Learning Representations*, 2022.

[74] Jeff Orkin. Three states and a plan: the ai of fear. In *Game developers conference*, volume 2006, page 4. CMP Game Group San Jose, California, 2006.

[75] Andreas Panayiotou, Theodoros Kyriakou, Marilena Lemonari, Yiorgos Chrysanthou, and Panayiotis Charalambous. Ccp: Configurable crowd profiles. In *ACM SIGGRAPH 2022 Conference Proceedings*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393379. doi: 10.1145/3528233.3530712. URL `https://doi.org/10.1145/3528233.3530712`.

[76] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*, 2021. URL `http://arxiv.org/abs/2006.07869`.

[77] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Social simulacra: Creating populated prototypes for social computing systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–18, 2022.

[78] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22, 2023.

[79] Tim Pearce and Jun Zhu. Counter-strike deathmatch with large-scale behavioural cloning. In *2022 IEEE Conference on Games (CoG)*, pages 104–111. IEEE, 2022.

[80] Stefano Pellegrini, Andreas Ess, Konrad Schindler, and Luc Van Gool. You'll never walk alone: Modeling social behavior for multi-target tracking. In *2009 IEEE 12th international conference on computer vision*, pages 261–268. IEEE, 2009.

[81] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[82] Xavier Puig, Eric Undersander, Andrew Szot, Mikael Dallaire Cote, Tsung-Yen Yang, Ruslan Partsey, Ruta Desai, Alexander Clegg, Michal Hlavac, So Yeon Min, Vladimír Vondruš, Theophile Gervet, Vincent-Pierre Berges, John M Turner, Oleksandr Maksymets, Zsolt Kira, Mrinal Kalakrishnan, Jitendra Malik, Devendra Singh Chaplot, Unnat Jain, Dhruv Batra, Akshara Rai, and Roozbeh Mottaghi. Habitat 3.0: A co-habitat for humans, avatars, and robots. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=4znwzG92CE`.

[83] PyTorch. Torchscript for deployment - pytorch tutorials 2.3.0+cu121 documentation, 2024. URL `https://pytorch.org/tutorials/recipes/torchscript_inference.html`.

[84] Steven Rabin. *Game AI pro: collected wisdom of game AI professionals*. CRC Press, 2013.

[85] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[86] Brent Randall. Valorant's 128-ticf servers — riot games technology, August 2020. URL `https://technology.riotgames.com/news/valorants-128-tick-servers`.

[87] Refrag. Restrat - how to watch cs:go demos with my team — refrag.gg - the ultimate cs2 training tool, March 2023. URL `https://wiki.refrag.gg/en/RESTRAT`.

[88] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, aug 1987. ISSN 0097-8930. doi: 10.1145/37402.37406. URL `https://doi.org/10.1145/37402.37406`.

[89] Matteo G Richiardi. The future of agent-based modeling. *Eastern Economic Journal*, 43(2): 271–287, 2017.

[90] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[91] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[92] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[93] Ari Seff, Brian Cera, Dian Chen, Mason Ng, Aurick Zhou, Nigamaa Nayakanti, Khaled S Refaat, Rami Al-Rfou, and Benjamin Sapp. Motionlm: Multi-agent motion forecasting as language modeling. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8579–8590, 2023.

[94] Seokin Seo, HyeongJoo Hwang, Hongseok Yang, and Kee-Eung Kim. Regularized behavior cloning for blocking the leakage of past action information. In *The 37th Conference on Neural Information Processing Systems (NeurIPS 2023)*. Neural information processing systems foundation, 2023.

[95] Alessandro Sestini, Joakim Bergdahl, Konrad Tollmar, Andrew D Bagdanov, and Linus Gisslén. Towards informed design and validation assistance in computer games using imitation learning. In *2023 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2023.

[96] Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.*, 42(4), jul 2023. ISSN 0730-0301. doi: 10.1145/3592427. URL `https://doi.org/10.1145/3592427`.

[97] Ben Sillis. Counter–strike's dust 2: The history of the map, July 2020. URL `https://www.redbull.com/us-en/counter-strike-dust-2-map-complete-history`.

[98] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676), 2017.

[99] Rich Stanton. Counter-strike: Global offensive removes bots from competitive — pc gamer, January 2021. URL `https://www.pcgamer.com/counter-strike-global-offensive-removes-bots-from-competitive/`.

[100] SourceMod Dev Team. Sourcemod: Half-life 2 scripting, March 2023. URL `https://www.sourcemod.net/`.

[101] The Fortnite Team. Fortnite matchmaking update - battle royale, September 2019. URL `https://www.fortnite.com/news/fortnite-matchmaking-update-battle-royale`.

[102] Tommy Thompson. How forza's drivatar actually works, June 2021. URL `https://www.gamedeveloper.com/design/how-forza-s-drivatar-actually-works`.

[103] Tommy Thompson. The ai of doom (1993), May 2022. URL `https://www.gamedeveloper.com/game-platforms/the-ai-of-doom-1993`.

[104] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[105] Arun Venkatraman, Byron Boots, Martial Hebert, and J Andrew Bagnell. Data as demonstrator with applications to system identification. In *ALR Workshop, NIPS*, 2014.

[106] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL `https://openreview.net/forum?id=ehfRiF0R3a`.

[107] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[108] Erica Weng, Hana Hoshino, Deva Ramanan, and Kris Kitani. Joint metrics matter: A better standard for trajectory forecasting. *arXiv preprint arXiv:2305.06292*, 2023.

[109] David Wolinski, Ming C. Lin, and Julien Pettré. Warpdriver: context-aware probabilistic motion prediction for crowd simulation. *ACM Trans. Graph.*, 35(6), dec 2016. ISSN 0730-0301. doi: 10.1145/2980179.2982442. URL `https://doi.org/10.1145/2980179.2982442`.

[110] Peter Xenopoulos and Claudio Silva. Esta: An esports trajectory and action dataset. *arXiv preprint arXiv:2209.09861*, 2022.

[111] Wesley Yin-Poole. Call of duty has 90 million players, half of all engagement on mobile, 2023. URL `https://www.ign.com/articles/call-of-duty-has-90-million-players-half-of-all-engagement-on-mobile`.

[112] Ye Yuan, Xinshuo Weng, Yanglan Ou, and Kris M Kitani. Agentformer: Agent-aware transformers for socio-temporal multi-agent forecasting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9813–9823, 2021.

[113] Hongxin Zhang, Zeyuan Wang, Qiushi Lyu, Zheyuan Zhang, Sunli Chen, Tianmin Shu, Yilun Du, and Chuang Gan. Combo: Compositional world models for embodied multi-agent cooperation. *arXiv preprint arXiv:2404.10775*, 2024.

[114] Duo Zheng, Shijia Huang, Lin Zhao, Yiwu Zhong, and Liwei Wang. Towards learning a generalist model for embodied navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13624–13634, 2024.