A Memory-Driven Action Selection Framework for Scalable Ambient NPC Behavior

by

Eric Buitron Lopez

Graduate Program in Computer Science

A project report submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Supervised by Dr. Roberto Solis-Oba

Acknowledgements

This report is the culmination of my master's project, and several people and institutions contributed in different ways to help me during my research. I would like to express my gratitude to Western University and the Computer Science department for allowing me to be part of their community and providing funding throughout my studies. I also thank Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI, México) for its financial support for maintenance through the "Becas de Posgrado para Maestrías y Doctorado en Ciencias y Humanidades en el Extranjero 2024" fellowship (CVU: 2021550).

Special thanks to my supervisor, Dr. Roberto Solis-Oba, for his guidance throughout this project, his thorough and detailed feedback on every chapter of this report, and his patience during the many rounds of revisions. His perspective consistently pushed me to make the writing clearer and more accessible to a broader audience. I also want to thank Dr. Mike Katchabaw for his availability and suggestions during the development of the demonstrations of this project.

I am grateful to my parents, Luz Maria and German, for always being there and supporting me through every step of this journey. Thanks for always encouraging me to do my best and providing guidance when needed. I also want to thank my sister, Isa, who made my arrival to a new country and university far less intimidating. Finally, and most importantly, special thanks to Pao for always believing in me and supporting me throughout this entire process.

Abstract

Ambient non-player characters (NPCs) in large open-world games must exhibit varied and context-appropriate behavior across large populations while operating under strict computational budgets. Existing approaches for managing NPCs face a fundamental trade-off: Planning and utility-based systems can produce diverse behavior but incur significant computational cost or require extensive parameter tuning, while lightweight reactive methods such as finite state machines and behavior trees scale well but frequently produce repetitive patterns. Furthermore, the most comprehensive existing framework targeting ambient NPC behavior (called CIVIL AI) is tightly coupled to a single game engine, limiting its broader applicability.

We present a scalable framework for ambient NPC behavior that addresses this trade-off through four contributions. First, a memory-driven action selection algorithm in which each NPC maintains a bounded record of recent decisions and favors untried or least-recently used options, producing behavioral variety without requiring planning, utility scoring, or per-character scripting. NPC behaviors are organized as directed graphs of actions that define the possible behavioral paths each character may follow. Second, a unified mechanism for evaluating conditions and modifying entity state across different sources of information, enabling NPC behavior to be defined independently of how a specific game organizes its internal data. Third, integrated behavioral continuity mechanisms combining interruption handling with context preservation, automatic fallback activation, and a failure safeguard that together keep NPCs active and behaving plausibly when unexpected conditions arise. Fourth, an engine-agnostic, data-driven implementation as a C++ dynamic-link library (DLL) with JSON-defined behaviors, separating decision-making logic from engine-specific execution systems.

We evaluated the framework through three approaches: A qualitative behavioral demonstration in Unity (30 NPCs across three character types in an ancient marketplace scenario), a cross-engine validation deploying the same compiled DLL in Unreal Engine, and quantitative performance testing scaling from 50 to 200 NPCs. The results show observable behavioral variety across NPCs sharing identical definitions, cross-engine portability, and sub-linear scaling with framework overhead reaching 3.41% of the frame budget at 200 NPCs, well below the 10% threshold commonly allocated to AI in commercial games. These results indicate that memory-driven action selection provides a practical and scalable approach for controlling large populations of ambient NPCs across different game engines in modern game environments.

Keywords: non-player characters, game AI, behavioral variety, action selection, engine-agnostic, ambient behavior, memory-driven, data-driven

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Research Objectives . . . 3
- 1.4 Contributions . . . 4
- 1.5 Basic Concepts and Definitions . . . 5
  - 1.5.1 Frame Budgets in Real-Time Games . . . 5
  - 1.5.2 AI in Video Games . . . 5
  - 1.5.3 Types of Characters in Video Games . . . 6
  - 1.5.4 Video Game Engines . . . 6
  - 1.5.5 Video Game Genres . . . 7
  - 1.5.6 What is realistic behavior? . . . 8
- 1.6 Report Organization . . . 9

2 Related Works **10**
- 2.1 NPC Behavior Algorithms . . . 10
  - 2.1.1 State-Based Algorithms . . . 10
  - 2.1.2 Decision-Making Algorithms . . . 11
  - 2.1.3 Planning-Based Algorithms . . . 14
  - 2.1.4 Schedule Systems . . . 17
  - 2.1.5 Hierarchical and Hybrid Approaches . . . 17
- 2.2 Supporting Systems for NPC Decision-Making . . . 18
  - 2.2.1 Knowledge Representation . . . 18
  - 2.2.2 Environmental Awareness . . . 19
- 2.3 Algorithm Optimization Techniques . . . 20
  - 2.3.1 Level of Detail (LoD) . . . 20
  - 2.3.2 Crowd Systems and Spatial Management . . . 20
  - 2.3.3 AI Frame Budget Practices in Commercial Games . . . 20
- 2.4 Implementation in Video Game Engines . . . 21
  - 2.4.1 Unity . . . 21
  - 2.4.2 Unreal Engine . . . 22
  - 2.4.3 Cross-Engine Observations . . . 23
- 2.5 Commercial Video Games Case Studies . . . 23
  - 2.5.1 Assassin's Creed Unity . . . 23
  - 2.5.2 The Sims . . . 24

2.5.3 Skyrim . . . 25
2.5.4 Red Dead Redemption . . . 25
2.5.5 Watch Dogs Legion . . . 26
2.5.6 Other Notable Implementations . . . 26
2.5.7 Case Study Observations . . . 27
2.6 NPC Behavior Research . . . 28
2.6.1 Behavioral Variety and Scalability . . . 28
2.6.2 Memory Representation . . . 29
2.6.3 Other Research Directions . . . 30

3 Framework Theoretical Model 32
3.1 Framework Components . . . 32
3.1.1 Entities . . . 32
3.1.2 State Operations . . . 33
3.1.3 Actions . . . 35
3.1.4 Behavior Sequences . . . 37
3.1.5 Memory System . . . 39
3.2 Conceptual Architecture . . . 42
3.2.1 Component Groups and Coordination . . . 43
3.2.2 Control Flow . . . 44
3.3 Decision-Making Process . . . 44
3.3.1 Sequence Lifecycle States . . . 44
3.3.2 Sequence Execution Process . . . 45
3.3.3 Action Execution Process . . . 47
3.3.4 Interruption Handling . . . 48
3.3.5 Resumption . . . 50
3.3.6 Behavioral Resilience and Fallbacks . . . 51
3.4 Framework Scope and Integration Requirements . . . 52

4 Framework Implementation 55
4.1 Implementation Design . . . 55
4.1.1 Design Goals . . . 55
4.1.2 Technology Selection Rationale . . . 55
4.2 Data Structures and Algorithms . . . 56
4.2.1 Entity Group . . . 57
4.2.2 Behavior Group . . . 58
4.2.3 Memory Group . . . 61
4.2.4 Configuration System . . . 63
4.3 DLL Architecture . . . 64
4.3.1 Service Layers . . . 66
4.3.2 Service Layer Coordination . . . 68
4.4 Framework Orchestration and Integration . . . 68
4.4.1 Framework Orchestration . . . 68
4.4.2 External C API . . . 70
4.4.3 Engine Integration Interfaces . . . 70

5 Integration with Game Engines **72**
5.1 Unity Integration . . . 72
5.1.1 Engine Overview . . . 72
5.1.2 Wrapper Layer Architecture . . . 73
5.1.3 Marketplace Demonstration . . . 74
5.1.4 Unity Editor Configuration . . . 75
5.2 Unreal Engine Integration . . . 78
5.2.1 Engine Overview . . . 78
5.2.2 Wrapper Layer Architecture . . . 78
5.2.3 Dance Club Demonstration . . . 79
5.2.4 Unreal Engine Editor Configuration . . . 79
5.3 Integration Comparison . . . 81

6 Evaluation **83**
6.1 Behavioral Demonstration . . . 83
6.2 Cross-Engine Validation . . . 85
6.3 Performance Evaluation . . . 86
6.3.1 Evaluation Methodology . . . 86
6.3.2 Test Protocol . . . 86
6.3.3 Timing Results . . . 87
6.3.4 Variable Workload Analysis . . . 89
6.3.5 Internal Performance Breakdown . . . 90
6.4 Evaluation Summary . . . 91

7 Conclusions **93**
7.1 Summary of Contributions . . . 93
7.2 Positioning Among Existing Solutions . . . 94
7.3 Limitations and Future Work . . . 96
7.4 Final Remarks . . . 98

Bibliography **99**

A Configuration File Specifications **104**
A.1 Configuration Files Dependencies . . . 104
A.2 State Schema Format . . . 104
A.3 Environmental Conditions Format . . . 105
A.4 Action Definitions Format . . . 106
A.5 Behavior Sequences Definition Format . . . 110
A.6 Entity Configuration Format . . . 113
A.6.1 Framework Entities . . . 114
A.6.2 Behavioral Entities . . . 115

B Supplementary Material **117**

1. Introduction

1.1 Motivation

Modern video games increasingly adopt open-world designs that give players the freedom to explore expansive game environments at their own pace. Unlike linear games where players progress through a predetermined sequence of locations, open-world games allow players to move freely through cities, landscapes, and other environments that must feel alive and inhabited regardless of where the player chooses to go or what they choose to do. Creating this illusion of a living world is one of the central challenges in open-world game development.

A critical element in achieving this illusion is the population of background characters that inhabit the game world. In video games, characters not controlled by human players are called non-player characters (NPCs). Among these, ambient NPCs are the characters that exist primarily to make the game world feel populated and believable: the pedestrians walking through a city, the vendors managing their shops, the villagers going about their daily routines. Unlike main NPCs who drive a game's narrative or serve as direct opponents, ambient NPCs exist in the background of a game. Players may never interact with most of these characters directly, but their presence and behavior shape the player's perception of the game world as a coherent, living place. A more detailed categorization of character types in video games is presented in Section 1.5.3.

Despite their importance to player immersion, ambient NPCs face a unique set of constraints that distinguish them from other types of game characters. They must exist at scale, as modern open-world games may need hundreds or even thousands of ambient characters visible simultaneously, while operating within strict computational budgets [27]. Game hardware resources are primarily allocated to rendering, physics, and player character systems, leaving ambient NPC behavior to function with a fraction of the available processing power (the concept of frame budgets and their implications for NPC systems is described in Section 1.5.1). At the same time, players have growing expectations for character behavior: they notice when every pedestrian follows the same walking pattern, when a vendor continues working outside during a storm, or when an NPC they encountered minutes ago shows no recognition of a previous interaction [47].

Commercial titles have attempted various approaches to address these challenges, ranging from scheduling daily routines to procedurally generating character activities, but limitations in behavioral variety persist across these implementations. A detailed analysis of NPC commercial approaches and their trade-offs is presented in Section 2.5.

This tension between behavioral quality and computational scalability motivates our research. We seek an approach that can produce behavioral variety and environmental awareness while remaining efficient enough to support hundreds of ambient characters.

1.2 Problem Statement

Our analysis of ambient NPC systems in commercial games (Section 2.5) and current research (Section 2.6) reveals three primary areas where existing approaches fall short of player expectations:

1. **Repetitive behavior and poor persistence**: Ambient NPCs tend to repeat the same actions in the same order, creating patterns that players recognize after extended observation. While some games have begun to address this (e.g. *Red Dead Redemption 2* implements a basic memory system for player interactions [53]), the behavioral variety of ambient NPCs remains limited and inconsistent. NPCs frequently lack any awareness of their own past actions, leading to behaviors that feel mechanical rather than lifelike.

2. **Poor environmental awareness**: Ambient NPCs often demonstrate limited understanding of the conditions around them when deciding what to do next. Characters may continue outdoor activities during storms, ignore significant events occurring nearby, or perform actions that are inappropriate for the time of day or their current location. This lack of context-awareness stems from the difficulty of incorporating environmental information into NPC decision-making without incurring substantial computational costs for each character.

3. **Limited social dynamics**: Most ambient NPCs interact only with the game environment (sitting on benches, opening doors) or with the player character. Meaningful interactions between ambient characters, such as group activities, relationship-influenced behavior, or realistic information sharing, are rare in commercial games and remain an active area of research (Section 2.6.3).

These limitations persist in part because existing behavior algorithms involve trade-offs that make them difficult to apply comprehensively to ambient NPC populations. Planning algorithms like Goal-Oriented Action Planning (GOAP) and Hierarchical Task Networks (HTN) generate varied action sequences but are too computationally expensive for hundreds of characters (Section 2.1.3). Utility-based systems produce dynamic, responsive behavior but require careful tuning of evaluation functions and have higher computational overhead than simpler approaches (Section 2.1.2). Schedule and state-based systems are efficient but inherently repetitive (Section 2.1.1 and Section 2.1.4). Furthermore, the most comprehensive existing framework specifically targeting ambient NPC behavior (CIVIL AI, described in Section 2.4.1) is tightly coupled to a single game engine, limiting its applicability across the broader game development industry.

Beyond algorithmic constraints, existing approaches typically require developers to write specialized decision-making code or carefully tune parameters for each character type. This development overhead further limits the behavioral variety that teams can achieve within production timelines.

Our research addresses the first two limitations directly (behavioral variety and context-aware action selection) while establishing an architectural foundation that could support

social dynamics in future extensions. We focus on these two areas because they represent the most fundamental gaps between player expectations and current ambient NPC capabilities, and because addressing them effectively is a prerequisite for meaningful social behavior.

1.3 Research Objectives

Based on the limitations identified above and the analysis of existing approaches presented in Chapter 2, we define the following research objectives:

1. **Enable behavioral variety without per-character scripting**. Ambient NPCs that share the same set of possible actions should make different choices over time. A group of vendor characters, for example, should not perform the same actions in the same order simply because they are all vendors. This variety should emerge from the framework's decision-making process rather than requiring developers to write unique logic for each character.

2. **Support context-aware action selection**. NPC decision-making should incorporate information about the current state of a game, such as the time of day, weather conditions, or the availability of nearby objects. Characters should select actions that are appropriate to their current circumstances rather than follow fixed patterns regardless of what is happening around them.

3. **Maintain behavioral continuity through disruptions**. When game events interrupt an NPC's current activity, or when game conditions prevent an NPC from continuing what it was doing, the system should keep the character active and behaving plausibly. Characters should not freeze, enter undefined states, or abruptly switch to unrelated activities when something unexpected occurs. The system should also prevent degraded states where characters become stuck or cycle through failed behaviors indefinitely.

4. **Provide an engine-agnostic, data-driven design**. The framework's behavioral logic should be separable from any specific game engine, allowing the same decision-making system to operate in different engines without modification. This separation ensures that the framework is not limited to a single engine's ecosystem and can be adopted across a broader range of game development projects.

5. **Achieve computational efficiency for large populations**. The decision-making overhead of the framework should consume only a small fraction of the total processing time available when running a game, enabling hundreds of characters to operate simultaneously without degrading game performance. Since ambient NPCs perform routine activities rather than computationally intensive decision-making, the framework should leave the majority of processing time available for rendering, physics, animation, and other game systems.

These objectives guide the design of our proposed framework, whose specific contributions are described below.

1.4 Contributions

1. **A memory-driven action selection algorithm for ambient NPCs**. We propose a selection mechanism in which each NPC maintains a record of its recent decisions and uses this history to guide future choices toward variety. NPC behaviors are organized as directed graphs of actions (called **behavior sequences** in our framework), which define the possible behavioral paths an NPC may follow. When an NPC reaches a decision point in a sequence where multiple valid paths exist, or when multiple entities satisfy the conditions for an action, the selection algorithm favors options the NPC has not tried recently. This produces varied behavior over time, meaning that characters of the same type (e.g., multiple vendors who can all perform the same set of actions) will naturally make different choices without developers needing to write unique decision logic for each character. Existing approaches to ambient NPC behavior either produce repetitive actions (state machines, schedule systems) or achieve variety at high computational cost (utility systems, planning algorithms). The memory-driven selection algorithm occupies a space between these extremes: it produces emergent variety through simple historical tracking rather than complex scoring or search.

2. **A unified mechanism for condition evaluation and property modification**. NPCs need to check whether an action makes sense before performing it (is this bench available? Is the market currently open? Does this character have enough energy?) and update the game world when actions occur (mark the bench as occupied, reduce the character's energy). These checks and modifications involve different sources of information: the NPC itself, other objects in the game, and environmental conditions. The framework provides a single, consistent mechanism, the **state operation**, for handling all of these, so that NPC behavior can be defined without depending on how a specific game organizes its internal data. Existing systems typically use separate approaches for different sources of information, such as smart objects for interactive elements, environmental queries for world conditions, and internal state checks for character properties. The unified state operation approach enables NPC behavior to be defined once and applied across different games and engines.

3. **Resilient behavioral continuity mechanisms**. We propose systems that work together to keep NPCs active and behaving appropriately even when things go wrong. When a game event interrupts an NPC mid-action, the framework preserves what the character was doing (for example, a vendor who was midway through arranging goods at their stall), how far along it was in its current activity, and which object it was interacting with, so that the interrupted activity can resume when conditions permit. If resumption is not possible, a fallback system automatically activates a safe default behavior, such as idle wandering, that keeps the character active. A safeguard mechanism prevents infinite loops if a character repeatedly fails to complete its assigned behaviors. While individual elements of interruption handling exist in commercial games, the combination of context preservation, automatic fallback activation, and failure safeguards as a built-in framework feature for ambient NPCs has not been previously proposed as an integrated mechanism.

4. **An engine-agnostic implementation with cross-engine validation**. We implement the framework as a C++ dynamic-link library (DLL) that separates all behavioral decision-making from game engine specifics. All definitions of what NPCs can do, including their available actions, the conditions under which each action is valid, and how actions affect the game world, are specified in external configuration files rather than engine-specific code. Integration with both Unity and Unreal Engine validates that the same behavioral logic and configuration files produce equivalent character behaviors across different engines, with developers only needing to provide a small integration layer that connects the framework with game engine systems. This architecture addresses the limitation identified in existing frameworks, such as CIVIL AI, which are tightly coupled to a single game engine.

1.5 Basic Concepts and Definitions

This section describes concepts and terminology that are used throughout the rest of this report. Throughout this report, we use the term **game world** to refer to the simulated environment that players navigate and characters inhabit, as distinct from the game's broader systems such as user interface, input handling, and scoring mechanics.

1.5.1 Frame Budgets in Real-Time Games

Real-time games operate under strict timing constraints. Each frame of the game must complete all processing within a fixed time budget determined by the target frame rate. At 60 frames per second (FPS), this budget is approximately 16.67 milliseconds per frame; at 30 FPS, it expands to approximately 33.33 milliseconds. The total budget is divided among competing subsystems including rendering, physics, animation, audio, and AI. Rendering typically consumes the largest share, and the remaining time must be distributed across all other systems. The computational cost of any AI system must therefore be evaluated not in isolation, but as a fraction of the total frame budget available for all non-rendering processing.

1.5.2 AI in Video Games

The term "AI" in video games refers to something different from its usage in academic research. In academia, AI generally refers to systems that can reason, learn, and solve problems, often approaching or exceeding human capabilities. These systems frequently require substantial computational resources for training and inference and may need specialized hardware. In video games, AI is one of many systems that must coexist and function within the frame budget constraints described in the previous section, where rendering consumes the majority of available processing time. For this reason, game AI tends to rely on ad-hoc solutions, heuristics, and lightweight algorithms rather than computationally expensive learning or reasoning systems [37].

Game AI can be divided into three categories: movement (how characters navigate through a game world), behavior (how characters decide what to do), and strategy (how characters plan over longer time horizons) [37]. This research focuses on behavioral algorithms, which determine the actions that characters perform in a game.

1.5.3 Types of Characters in Video Games

Characters in video games can be categorized based on their role and level of interactivity. Player characters are controlled directly by human players. Non-player characters (NPCs) are any characters in a game that are not controlled by human players. NPCs can be further categorized based on their significance and functionality. Main NPCs have significant narrative or gameplay importance and tend to have substantial interactions with the player. Ambient NPCs populate the game world and create a sense of realism and immersion but tend to have limited interactions with or significance to the player.

This research focuses on ambient NPCs, which typically receive fewer computational resources than other character types in a game's AI systems. This resource disparity is one reason why ambient NPCs often exhibit unrealistic or repetitive behavior that can break player immersion. Ambient NPCs also present unique challenges because they must exist in much larger numbers (hundreds or even thousands) than other NPCs while operating under strict hardware resource constraints. Key considerations when designing ambient NPC systems are therefore performance, scalability, and ease of implementation [27].

1.5.4 Video Game Engines

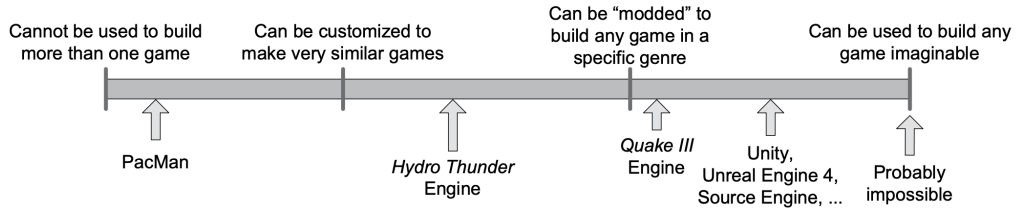

Figure 1.1: Reusability of a game engine. Taken from [28].

A video game engine is the software architecture and collection of components used to develop a video game. Engines typically include core systems such as graphics rendering, physics simulation, audio management, input handling, and asset management (e.g., 3D models and textures). They also provide development tools and cross-platform APIs that allow developers to focus on game-specific logic rather than the low-level programming required for each platform on which the game will run. The goal of a game engine is to provide reusable software that can support many different games without requiring extensive modifications. In practice, the degree of reusability varies: Engines optimized for a specific game type or platform may be difficult to adapt for different purposes, while more general engines sacrifice

some efficiency for broader applicability [28]. Figure 1.1 illustrates this trade-off between specialization and generality in game engine design.

Game engines can be broadly categorized into three types. Commercial engines are developed and maintained by companies focused on distributing their software. Some of the most popular commercial engines are Unreal Engine and Unity. These engines offer broad functionality and professional support but come with licensing costs and a general-purpose design that may be less efficient for specific game types. Open-source engines, such as Godot, are free and allow developers to modify the engine itself, but typically have less professional support. Proprietary engines are built and maintained by game studios for their own titles. Examples include the Rockstar Advanced Game Engine (RAGE) used for games such as *Red Dead Redemption* and *Grand Theft Auto*, and the Creation Engine used for games like *Skyrim* and *Fallout 4*. These engines can be optimized for a studio's specific needs but require significant resources to maintain, which is one reason why several studios have transitioned to commercial engines [21].

The 2026 State of the Game Industry survey conducted by the Game Developers Conference (GDC) shows that Unreal Engine and Unity are the most widely used engines, with Unreal Engine being used by 42% and Unity by 30% of surveyed developers (Figure 1.2) [23]. This landscape is relevant to our research because a main goal is to design a framework that can be integrated across different game engines. The dominance of Unity and Unreal Engine in the industry informed our decision to validate our framework in both engines (Chapter 5).

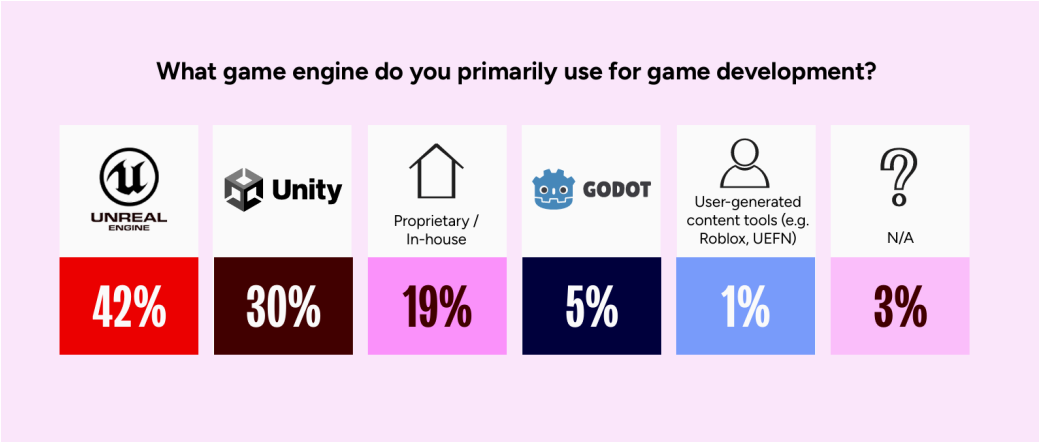

Figure 1.2: State of game engines used for game development in 2026. Taken from [23].

1.5.5 Video Game Genres

Video games span many genres, each with different requirements for ambient NPC behavior. This section briefly describes the genres that appear in the commercial case studies analyzed in Section 2.5.

Open-world games give players the freedom to explore the entire game world rather than progressing through a linear sequence of locations. The important aspect is the freedom that the player has while playing rather than the actual size of the game world. Maintaining

the illusion of a living world as players explore freely is one of the primary challenges that ambient NPCs address.

Action/adventure games combine exploration and narrative progression with challenges and combat opportunities. These games balance moment-to-moment gameplay with long-term story progression. Role-playing games (RPGs) emphasize character development and player choices that affect the game's story. Shooter games focus on weapon-based combat with the goal of eliminating enemies, managing resources, and navigating the game environment. Simulation games attempt to recreate real-world activities, systems, or experiences, and are known for having interconnected gameplay systems. These games range from city-building and vehicle operation to simulating the daily lives of characters. Stealth games task players with navigating through areas while avoiding detection, sometimes offering combat as an alternative but rewarding undetected completion of objectives.

Modern video games frequently combine elements from several genres, creating hybrid experiences. In recent years, many games have adopted open-world structures alongside their primary genre, which increases the importance of incorporating believable ambient NPCs. The case studies in Section 2.5 explore this trend and its implications for ambient NPC behavior.

1.5.6 What is realistic behavior?

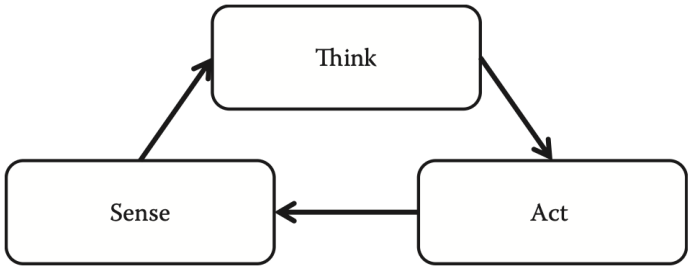

Figure 1.3: Sense-Think-Act Model. Taken from [11].

It is important to define what is meant by realistic behavior of characters. In the context of video games, realistic behavior means that NPCs act in ways that align with human expectations of how real people or other creatures would behave in similar circumstances. Characters should demonstrate both reactivity and deliberation: They should be responsive when perceiving changes in the game world and should have the ability to make decisions and engage in consequent actions. This is known as the sense-think-act model [11] shown in Figure 1.3.

The sense-think-act model applies to all characters in a game; however, our focus is on ambient NPCs. As described in Section 1.5.3, these characters create the illusion of life within the game, but because they are not given high priority in a game's AI systems, they are more likely to cause breaks in realism. Sunshine-Hill [47] identifies three types of realism breaks that players notice: An unrealistic state (when the observable simulation is

wrong), fundamental discontinuity (when the current state of a game is incompatible with what happened in the past), and unrealistic long-term behavior (when an extended period of observation reveals incorrect behavior).

The goal of realistic ambient NPC behavior can therefore be understood as creating believable, varied, and contextually appropriate character actions that enhance the player's immersion and convey the sense of a living world. These three qualities correspond directly to the research objectives defined in Section 1.3: behavioral variety (Objective 1), contextual appropriateness through environmental awareness (Objective 2), and believability through behavioral continuity (Objective 3).

1.6 Report Organization

The remainder of this report is organized as follows:

- **Chapter 2** analyzes existing behavior algorithms, knowledge representation systems, commercial game implementations, and recent research on ambient NPC behavior, establishing the technical context for our framework.
- **Chapter 3** presents the theoretical model of our framework, defining its components and decision-making processes at a conceptual level.
- **Chapter 4** presents the implementation of our framework, beginning with language-agnostic design requirements and data structure specifications, followed by our concrete realization as a C++ dynamic-link library with JSON configuration files.
- **Chapter 5** describes the integration of our implementation with Unity and Unreal Engine.
- **Chapter 6** presents observations on NPC behavior quality, cross-engine implementation validation, and experimental results evaluating our framework's performance.
- **Chapter 7** discusses conclusions, limitations, and future work.
- **Appendix A** provides a user manual for the configuration file format used to define NPC behaviors.
- **Appendix B** provides a link to supplementary review materials for the project.

2. Related Works

The limitations identified in Section 1.2 arise in part from the trade-offs inherent in existing behavior algorithms for NPCs. This chapter examines these algorithms and the broader technical landscape in which ambient NPC behavior operates. Section 2.1 describes the behavior algorithms used for NPC decision-making, evaluating each for its suitability for ambient NPC populations. Section 2.2 through Section 2.3 cover the supporting systems and optimization techniques that complement these algorithms. Section 2.5 examines how commercial video games implement ambient NPC behavior in practice. Section 2.6 reviews current research relevant to the contributions of this work.

2.1 NPC Behavior Algorithms

This section explains different algorithms used for controlling the behavior of NPCs. These algorithms can be organized into four categories: state-based, decision-making, planning-based, and time-based. While these algorithms are used for different types of characters in games, our focus is on their suitability for ambient NPCs. Section 2.1.5 explains that several of these algorithms have hierarchical properties and many of them are often combined in commercial games.

2.1.1 State-Based Algorithms

State-based algorithms represent NPC behavior as a collection of discrete states with defined transitions between them. A state defines what an NPC is currently doing (such as patrolling, eating, or sleeping), and a transition is a rule that determines when and how the NPC should change from one state to another. Transitions are typically triggered by conditions in a game world or by properties of an NPC [15, 37]. The core limitation of state-based algorithms for ambient NPCs is that they produce predictable, mechanical behavior patterns. Since transitions are typically deterministic, players observing an NPC over time will notice that it always follows the same behavioral patterns. Below we describe two popular state-based algorithms.

Finite State Machines (FSM)

A finite state machine (FSM) is one of the most widely used behavior models for NPCs. An NPC governed by an FSM can be in exactly one state at any given time, with transitions between states that are typically deterministic. Figure 2.1 shows an example of an FSM where a character transitions between patrolling, investigating, attacking, and fleeing states based on game conditions.

The conceptual simplicity of FSMs makes them easy to implement and debug, and their minimal computational overhead makes them suitable for controlling many characters simul-

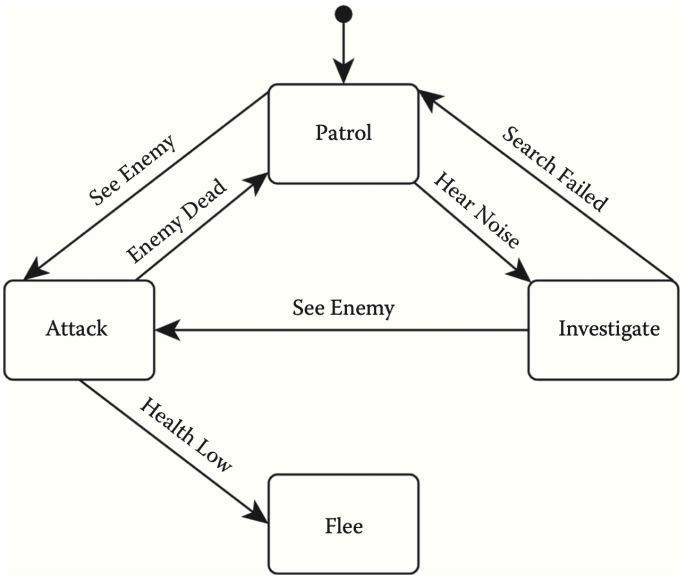

Figure 2.1: Example of an FSM diagram. Taken from [15].

taneously. FSM states can also map directly to animation states, which simplifies integration with animation systems. However, the number of states and transitions grows rapidly as behaviors become more complex, making FSMs difficult to maintain and extend. They are also difficult to reuse across different character types without significant modification. For ambient NPCs, FSMs work well for characters with limited interactions, such as pedestrians that follow basic routines or NPCs that respond to a small number of environmental conditions.

Hierarchical State Machines (HFSM)

Hierarchical state machines extend FSMs by allowing states to contain entire sub-state machines, creating a nested structure. This reduces the number of top-level states needed to represent complex behaviors and allows common sub-behaviors to be reused across different character types. However, HFSMs still face scalability issues as behavior complexity increases, and the nested structure adds implementation and debugging complexity compared to standard FSMs. For ambient NPCs, HFSMs are appropriate when characters exhibit distinct behavior modes (such as a shopkeeper alternating between serving customers, restocking shelves, and taking breaks), each with their own set of sub-behaviors.

2.1.2 Decision-Making Algorithms

Decision-making algorithms focus on selecting appropriate actions based on the current conditions of a game rather than transitioning between predefined states. They emphasize

the process of choosing what to do next, which makes them more flexible than state-based approaches but also more complex [15, 37].

Rule-based Systems

Rule-based systems use conditional logic (if-then rules) to determine NPC behavior based on the current game state. These systems consist of a collection of condition-action rules, a mechanism that evaluates which conditions are currently true, and a selection mechanism that determines which rule to follow when multiple conditions are satisfied.

Rule-based systems are intuitive for designers to create and modify, and they offer low computational overhead for simple rule sets. However, as the number of rules increases, the system becomes difficult to maintain due to unexpected interactions between rules. These systems also tend to create deterministic behaviors that become predictable for players. For ambient NPCs, rule-based systems work well for characters that need to respond consistently to specific environmental conditions, such as crowd behaviors where characters react to player actions or changes in time of day.

Utility Systems

Utility systems evaluate multiple possible actions by computing a numerical score (called utility) for each action in the current context, then selecting the action with the highest score. The utility of an action represents how desirable it is for the character to perform that action given the current game conditions. Each action defines a utility function that takes as inputs distance, character state (e.g. energy or hunger), time of day, and other relevant factors, and produces a score. A selection mechanism then chooses the highest-scoring action [26]. Figure 2.2 shows an example of how different inputs are combined to compute utility scores for two competing actions.

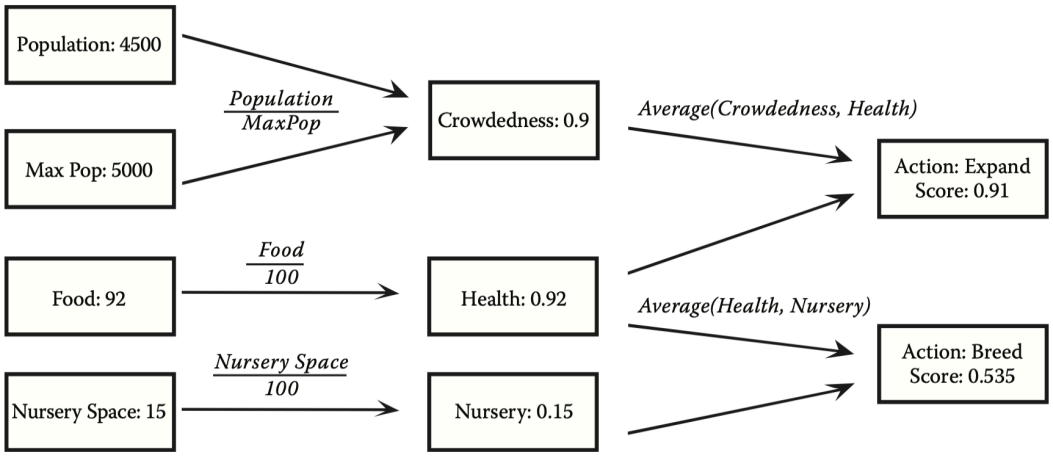

Figure 2.2: Example of combining inputs into a utility score. Taken from [26].

Utility systems can be extended with several mechanisms to improve the behaviors that they produce. Actions can be grouped into priority categories (called buckets), where the selection mechanism considers higher-priority groups first. An inertia mechanism can prevent characters from constantly switching actions by temporarily increasing the score of the currently active action. These extensions help produce more stable, believable behavior patterns.

Utility systems create more dynamic and responsive behavior than state-based or rule-based approaches because they naturally handle trade-offs between competing priorities and reduce behavioral predictability through weighted scoring. However, they require careful tuning of utility functions and weights for each action. This tuning complexity makes them more difficult to debug than deterministic systems, and they have higher computational overhead than simpler approaches. For ambient NPCs, utility systems are valuable for creating varied behavior among characters of the same type. Two shoppers in the same area might make different choices based on their individual utility calculations, making groups of identically defined NPCs appear more diverse. However, the tuning effort required for each action's utility function limits how many distinct behaviors a development team can create within production timelines.

Behavior Trees

Behavior trees organize decision-making into a hierarchical tree structure where execution flows from a root node down to actions at the leaf nodes. The tree is evaluated each frame to determine which actions an NPC should perform [15, 37]. Leaf nodes are either action nodes (which execute behaviors such as "move to position" or "play animation") or condition nodes (which check the game state, such as "is enemy nearby?"). Internal composite nodes control execution flow: sequence nodes process their children in order and fail if any child fails, selector nodes process children in order until one succeeds, and parallel nodes process multiple children simultaneously. Decorator nodes modify the behavior of a single child node. For example, a repeat decorator re-executes its child node a specified number of times, and an inverter decorator changes a child node's outcome from success to failure or vice versa. Figure 2.3 shows how these node types work together in a simple combat behavior.

Behavior trees are highly modular because subtrees can be designed as self-contained units and reused across different contexts. Their tree structure also provides a clear visual representation that designers can understand, and the separation between the tree's decision logic (composite and decorator nodes) and the actual behavior implementations (leaf nodes) keeps the architecture organized. However, behavior trees can be computationally expensive for deeply nested trees because the entire tree is traversed from root to active leaves every frame. They also often require specialized visual editors for designers to work with. A key limitation is that basic behavior trees are deterministic: they always evaluate nodes in the same order, producing the same behavior under the same conditions. Variation can be introduced by extending the node types to include probabilistic or utility-based selectors [35]. Behavior trees have become popular for ambient NPCs due to their balance of expressiveness and modularity, particularly for characters with moderately complex behaviors that need to be reused across many instances, such as citizens following daily routines while responding to dynamic events.

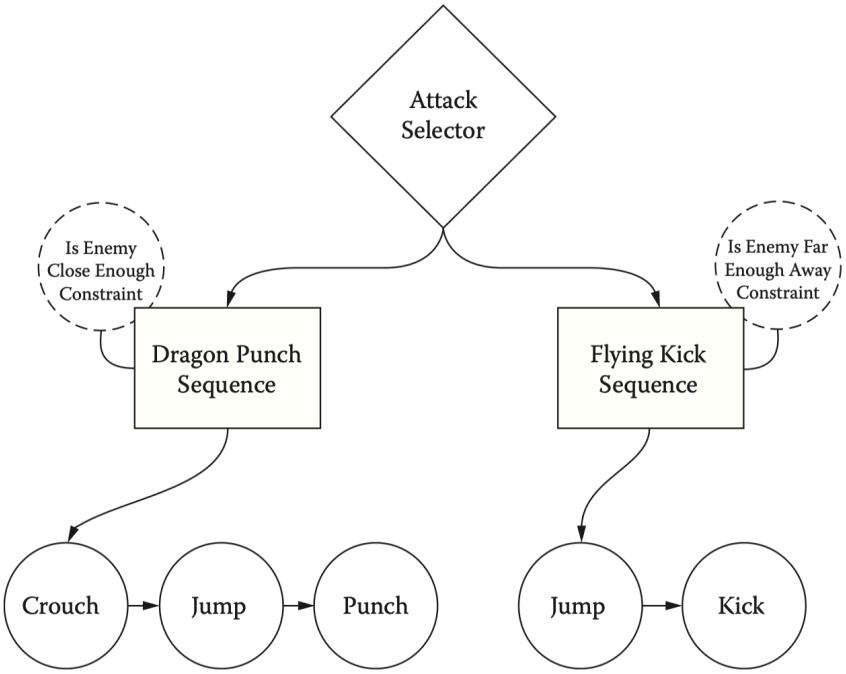

Figure 2.3: Example of a Behavior Tree. Taken from [29].

2.1.3 Planning-Based Algorithms

Planning-based algorithms specify sequences of actions to achieve specific goals rather than selecting only the next action. These algorithms evaluate potential action plans based on their ability to transition from the current state of a character to a desired goal state. Planning-based algorithms produce varied, context-sensitive behavior but at significantly higher computational cost than reactive methods [15, 37].

Goal Oriented Action Planning (GOAP)

GOAP is a planning system that determines a sequence of actions to transition from the current state of a character to a state that satisfies a specific goal. The system includes a set of actions with preconditions and effects, a symbolic representation of the current state, one or more goal states, and a planning algorithm that finds an action sequence through backwards chaining (starting from the desired goal state and working back to find the actions needed to reach it). Figure 2.4 illustrates this process for a character whose goal is to eliminate a target.

GOAP generates action sequences that adapt to the current game conditions and can produce different action plans as conditions change. However, GOAP has significantly higher computational overhead than reactive systems, and the preconditions and effects of actions must be carefully designed to avoid plans that appear illogical to players. For ambient NPCs, GOAP is generally too computationally expensive for large populations. GOAP may

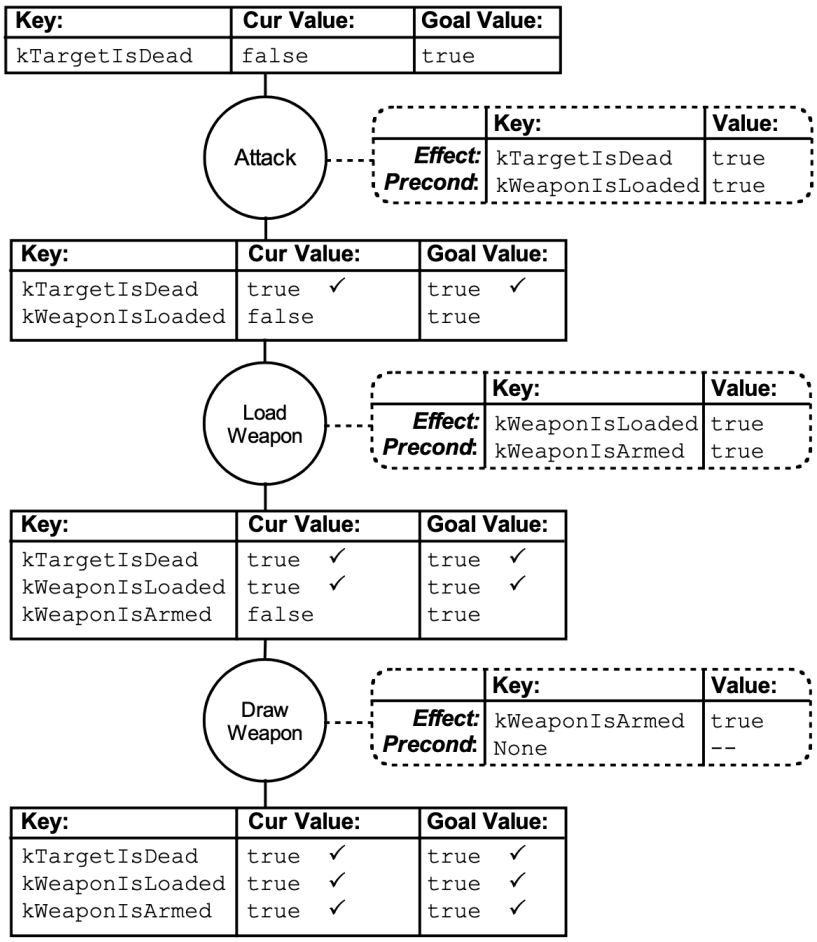

Figure 2.4: GOAP backwards planning example. Taken from [41].

be suitable for a small number of prominent ambient characters that need to demonstrate goal-directed behavior.

Cardon and Jacopin [9] addressed the computational cost of GOAP by parallelizing the planning process on a GPU, demonstrating the ability to plan the actions of thousands of NPCs per frame on a GTX 1080 graphics card. Their approach improves scalability through hardware acceleration but still relies on per-decision search over action sequences and requires GPU resources that may compete with rendering. This represents a different strategy for addressing the variety-scalability trade-off: accelerating the planning process rather than replacing it with a lighter-weight mechanism.

Hierarchical Task Networks (HTN)

HTN planning organizes actions into hierarchical tasks that can be decomposed into subtasks. The system includes primitive tasks, compound tasks, methods that specify how to

accomplish compound tasks, and a planning algorithm that recursively decomposes compound tasks into executable sequences of primitive tasks. Starting from a high-level goal (such as "defend base"), the planner decomposes compound tasks until it reaches primitive tasks that can be directly executed (such as "get weapon," "move to position," "attack"). If a decomposition fails because the current game state does not satisfy a task's preconditions, the planner backtracks and attempts an alternative decomposition of the same compound task. Figure 2.5 shows an overview of this system.

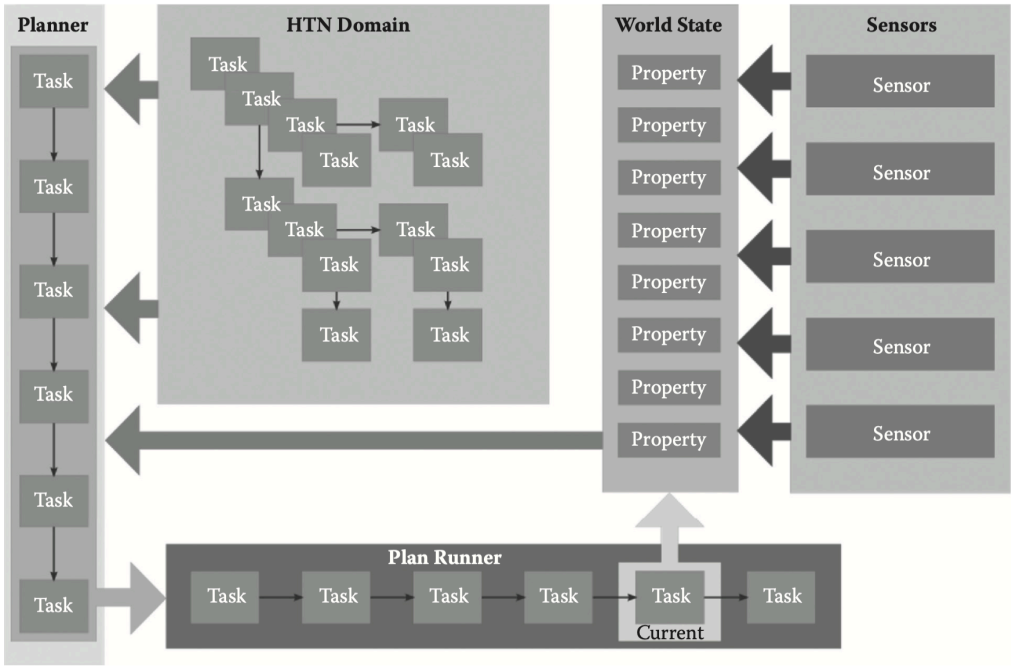

Figure 2.5: Overview of an HTN System. Taken from [30].

HTN planning offers more efficient planning than GOAP for well-structured domains because the predefined task hierarchy constrains the search space. It enables planning at multiple abstraction levels simultaneously and supports explicit precondition and effect modeling that allows the planner to anticipate how actions will change game state. This forward-looking capability distinguishes HTN from reactive hierarchical systems like behavior trees and HFSMs, which respond to current conditions rather than planning ahead. However, HTN systems are complex to implement and require significant upfront design of task hierarchies, making them less flexible than GOAP for handling unexpected situations. Like GOAP, HTN planning is computationally expensive for large ambient NPC populations, though it may be useful for ambient characters that perform complex, structured activities with multiple steps.

2.1.4 Schedule Systems

Schedule systems organize NPC behavior around game time, assigning different activities to different time periods. They typically include time blocks with assigned behaviors, transition logic between scheduled activities, priority systems for handling interruptions, and sometimes procedural schedule generation based on character roles.

Schedule systems create believable daily routines and can coordinate behaviors across multiple NPCs efficiently. Once established, they require relatively low computational overhead. However, the predictable patterns created by fixed schedules offer no behavioral variation: a character performing the same routine every day quickly feels mechanical to an observant player. It is also difficult to handle all cases of unexpected interruptions gracefully, and mixing scheduled behaviors with dynamic responses requires careful design. For ambient NPCs, schedule systems are commonly used in open-world role playing games and life simulation games to create the impression that characters have lives independent of the player. Their predictability is one of the primary areas where players lose immersion, as identified in Section 1.2.

2.1.5 Hierarchical and Hybrid Approaches

Several of the algorithms described above share a hierarchical nature: HFSMs, behavior trees, and HTN planning all organize behavior into abstraction levels where higher levels define broad behavioral categories and lower levels handle specific actions. This hierarchical organization offers benefits for ambient NPC behavior, including making behaviors scalable, accessible to designers, and providing natural points for introducing variation. It also enables performance optimization by updating high-level decisions less frequently than low-level execution. However, hierarchical approaches increase implementation and debugging complexity, can create rigid structures if the hierarchy is poorly designed, and introduce complexity in communication between levels [18].

In practice, commercial games rarely use a single behavior algorithm. Instead, they combine multiple approaches to leverage their respective strengths. Common combinations include FSMs or behavior trees with utility systems [35], behavior trees with planning or schedule systems, and HFSMs with rule systems. These hybrid approaches often organize NPC behavior into layers operating at different time scales: a strategic layer for long-term goal selection, a tactical layer for medium-term decision-making, and an execution layer for immediate actions. This layered approach allows each layer to use the algorithm best suited to its requirements.

The use of data-driven approaches has also recently increased, allowing designers to create behaviors from reusable components using visual tools rather than writing code. These systems typically combine elements from multiple algorithms and allow behavior modification without recompilation. However, most data-driven tools are tightly coupled to specific game engines, which limits their portability. The case studies in Section 2.5 illustrate several of these hybrid combinations in commercial games.

2.2 Supporting Systems for NPC Decision-Making

The behavior algorithms described in Section 2.1 determine what an NPC should do next, but they depend on supporting systems that provide NPCs with the information needed to make those decisions and that manage how NPCs interact with objects in the game world. This section describes these supporting systems, organized into knowledge representation approaches (how information about the game world is stored and accessed) and environmental awareness approaches (how NPCs perceive and gather information from their surroundings).

2.2.1 Knowledge Representation

Knowledge representation systems define how information about the game world is organized and made available to NPC decision-making algorithms. The choice of representation affects how NPCs access information about other characters, environmental conditions, and interactive objects. The three approaches described below are common in commercial games.

Blackboard systems provide a shared memory space that different components of an AI system can read from and write to. Data is typically organized as key-value pairs, and notification mechanisms can alert components when relevant data changes. For ambient NPCs, blackboards can facilitate coordinated behavior by sharing environmental information (such as weather conditions, time of day, or crowd density) across multiple characters without requiring direct communication between them. However, a centralized blackboard can create access bottlenecks when many NPCs query it simultaneously, and characters may receive outdated information if update mechanisms are inefficient[18].

Smart Objects embed interaction information directly within game world objects rather than within the NPCs themselves. Instead of each NPC containing logic for how to interact with every type of object, the objects declare what interactions they support, including required animations, NPC positions, and state changes. NPCs can query nearby objects to determine what interactions are available. This approach reduces the complexity of individual NPCs and creates a modular, reusable interaction system: developers can add new interactive objects without modifying NPC code. Smart objects are used extensively in *The Sims* series (Section 2.5.2), where objects advertise their interaction capabilities to characters. The concept of objects declaring their own supported interactions is relevant to the entity model presented in Chapter 3. However, smart objects can lead to repetitive behaviors if the variety of interactive objects in a game environment is limited, and coordinating multiple NPCs attempting to use the same object requires additional management logic [6, 18].

Influence maps represent spatial information as continuous values distributed across a game world, allowing NPCs to reason about locations and areas (such as "how dangerous or busy is this area?") rather than about discrete objects or states. Influence maps are implemented using grid-based or node-based data structures with propagation algorithms that spread influence values across space and decay mechanisms that age information over time [48]. Influence maps support spatially aware crowd behaviors and natural-looking activity

distribution, but they are memory intensive for large game worlds and computationally expensive to update. They are more commonly used for NPCs making strategic or movement decisions than for the routine behavioral decisions that ambient NPCs perform.

2.2.2 Environmental Awareness

NPC behavior algorithms need information about the game world to make contextually appropriate decisions. Below we describe some approaches that exist for providing this information, representing different trade-offs between realism and computational efficiency.

Sensory simulation systems model how NPCs gather information through virtual senses such as sight, hearing, and sometimes touch or smell. Vision systems use techniques such as raycasting or field-of-view calculations to determine what an NPC can see, while audio systems model sound propagation accounting for volume, distance, and obstacles. These systems create believable reactions to environmental changes and limit NPC knowledge to what they could realistically perceive, which supports gameplay mechanics like stealth. However, running sensory simulations for large numbers of ambient NPCs is computationally expensive, and the systems require careful tuning to balance perceptual realism with gameplay needs [42, 46].

Environmental query systems provide NPCs with direct access to game world information without simulating the process of perception. Rather than requiring an NPC to "look around" and potentially miss objects behind obstacles, an environmental query can directly retrieve information such as "which benches are within 50 meters?" regardless of visibility. This approach sacrifices perceptual realism for computational efficiency and implementation simplicity. Spatial queries can find nearby objects or characters, state queries can check environmental conditions or object properties, and caching mechanisms can reduce redundant queries. Environmental queries scale well to large NPC populations because they avoid the per-character cost of sensory simulation, making them a common choice for ambient NPCs [18].

Awareness representation systems model what NPCs know and remember about the game world over time. These systems typically include memory structures that store previously perceived information, certainty levels reflecting confidence in that knowledge, forgetting mechanisms that degrade information over time, and knowledge-sharing mechanisms between NPCs [42, 46]. Awareness systems enable more persistent NPC responses to events and are relevant to the memory system described in Chapter 3 of this report, though our framework's memory system serves a different purpose (tracking decision history for behavioral variety rather than modeling perceptual knowledge).

The environmental awareness approach chosen for a game significantly affects the believability and computational cost of ambient NPC behavior. Full sensory simulation produces the most realistic results but is impractical for large populations. Most commercial games use a combination of simplified environmental queries for ambient NPCs and more detailed sensory systems for a smaller number of important characters.

2.3 Algorithm Optimization Techniques

The behavior algorithms and supporting systems described in the previous sections must operate within the strict frame budget constraints discussed in Section 1.5.1. This section describes the techniques used to optimize NPC behavior for large populations, followed by an overview of how commercial games allocate frame budget to AI systems in practice.

2.3.1 Level of Detail (LoD)

Level of detail (LoD) systems adjust the complexity and update frequency of NPC behaviors based on factors such as distance from the player, visibility, and narrative importance. Implementing LoD requires creating multiple behavior models of varying complexity for each NPC type, specifying transition criteria for switching between detail levels, and using resource budgeting to allocate more processing to visible or important NPCs [47].

LoD techniques can substantially improve performance while maintaining believable behaviors where players are most likely to notice them. These techniques also enable larger NPC populations by reducing the average per-character cost. However, LoD requires additional development effort since multiple behavior systems must be created for each detail level, and poorly managed transitions between levels can create noticeable behavioral changes that break immersion.

2.3.2 Crowd Systems and Spatial Management

Crowd systems manage large groups of ambient NPCs that move and behave as a coordinated group rather than as fully independent characters. These systems typically use group-based movement algorithms (such as flocking and steering behaviors) and shared decision-making systems. Crowd systems improve the performance of large NPC populations and create realistic group movement, but they reduce the individuality of each NPC and make it challenging to transition between crowd and individual behaviors.

Spatial management techniques complement crowd systems by optimizing how NPCs are distributed and processed across a game world. These techniques include spatial partitioning for efficient NPC queries, population density controls based on area importance and player visibility, and spawn/despawn systems that add or remove NPCs based on player proximity. Spatial management creates efficient resource allocation across large game worlds but can create noticeable transitions when NPCs appear or disappear, and it limits the long-term persistence of NPCs across distant areas.

2.3.3 AI Frame Budget Practices in Commercial Games

While the optimization techniques described above (level of detail, crowd systems, and spatial management) address how commercial games reduce the cost of NPC behavior, it is also useful to consider how much of the frame budget commercial games allocate to AI in practice.

Exact allocations vary by genre, platform, and title, but several published data points help establish reasonable expectations.

In *Hitman Absolution*, IO Interactive supports up to 300 NPCs per level but only gives a full AI update to a subset of 40 to 120 each frame through time-slicing [16]. In *Assassin's Creed Unity*, Ubisoft limits full AI agents to 40 out of up to 10,000 rendered crowd NPCs [12]. Plch et al. reported that their AI system for a *Kingdom Come: Deliverance* consumed less than 1 ms with 30 complex NPCs and less than 2 ms with 300 simple NPCs, representing approximately 6–12% of the frame budget depending on the target frame rate [44, 45].

These examples indicate that commercial AI systems are designed to consume a small, bounded fraction of the total frame budget, typically through a combination of time-slicing, level-of-detail scaling, and strict per-frame limits. The available evidence suggests that production AI subsystems target roughly 5–10% of the frame budget. This observation will inform the performance target used to evaluate the proposed framework in Chapter 6.

2.4 Implementation in Video Game Engines

The algorithms and systems described in the previous sections are made available to developers through the AI tools and frameworks provided by game engines. This section describes the AI tooling in the two most widely used engines (Unity and Unreal Engine, as established in Section 1.5.4) and examines the patterns that emerge across engine implementations. CIVIL AI is described in greater detail than other tools because it is the only identified framework specifically targeting ambient NPC behavior, making it the most directly comparable system to the framework proposed in this report.

2.4.1 Unity

Unity's AI tooling consists of first-party packages and a marketplace of third-party solutions. The official packages have historically focused on navigation and pathfinding, though Unity recently released an official behavior trees package that provides a standardized implementation of this algorithm. While relatively basic compared to third-party alternatives, its inclusion signals that behavior trees are an industry standard for game AI.

The most relevant third-party solutions include Behavior Designer (a visual node-based behavior tree editor), NodeCanvas (a framework offering behavior trees, FSMs, and blackboard-based systems), and ML-Agents (a machine learning toolkit for training character behaviors). Unity's ecosystem is characterized by variety but also inconsistency: the reliance on third-party tools creates potential compatibility issues when combining multiple solutions within a single project.

CIVIL AI

CIVIL AI is a third-party framework created by Bard Tree Limited that specifically targets ambient NPC behavior in Unity. It integrates multiple behavior algorithms within a modular, layered architecture that includes utility-based decision-making, HTN planning,

and a schedule system. It also implements smart objects for object-driven interactions, a grid-based influence map for spatial awareness, a personality traits module that influences the weights of its utility system, and a memory system that tracks interactions and environmental changes. Actions in CIVIL AI have preconditions and effects, and the framework can handle interruptions. For performance optimization, it incorporates a three-level LoD system based on distance from the player, along with batch processing and instance pooling [51].

The framework provides a designer-accessible visual system that requires little to no programming for basic ambient character implementations. It supports several pre-built activities, can be extended with custom behaviors, supports environments that change during runtime, and is optimized for 50 to 200 simultaneous active NPCs on mid-range hardware. However, CIVIL AI has several limitations relevant to our goals. It has a significant per-character memory cost and requires considerable setup time. Most notably for our research, CIVIL AI has been designed exclusively for the Unity game engine and is tightly coupled with Unity's internal systems, which means that NPC behaviors created with CIVIL AI cannot be transferred to other engines. This engine coupling is a limitation that our framework's engine-agnostic architecture (Contribution 4) specifically addresses.

CIVIL AI is one of the most comprehensive implementations of a framework targeting ambient NPCs. It addresses many of the same concerns as our framework (behavioral variety through utility scoring, context-awareness through preconditions, and interruption handling) but takes a fundamentally different approach: it achieves variety through utility-based scoring functions that require careful tuning, while our framework achieves variety through memory-driven selection that requires no tuning. CIVIL AI's documentation does not describe integrated fallback mechanisms or failure safeguards comparable to those proposed in Contribution 3 of this report.

2.4.2 Unreal Engine

Unreal Engine takes a more integrated approach to its tooling, providing built-in systems rather than relying on third-party solutions. Unreal Engine's tools include a behavior tree system with integrated blackboard-based knowledge representation, StateTree (a system that combines HFSMs with behavior trees for flexible decision-making), an Environment Query System (EQS) for spatial and environmental awareness, a perception system supporting sight, hearing, and damage-based awareness, smart object support, and the MassEntity framework designed for managing large crowds or ambient NPC populations [20].

The integrated nature of Unreal Engine's AI systems means that tools work together consistently and are maintained by the engine developer. However, this integration comes with a steeper learning curve compared to Unity's modular approach, and developers have less flexibility to substitute alternative approaches for individual AI components. Like Unity's tools, all of Unreal Engine's AI systems are tightly coupled to the engine itself: behaviors created using Unreal's behavior trees or EQS cannot be transferred to other engines without reimplementation.

2.4.3 Cross-Engine Observations

Several patterns emerge from examining the AI tooling across game engines. Behavior trees are present in nearly every major engine, supporting their status as the most widely adopted algorithm for NPC behavior. Blackboard systems are the standard approach for knowledge representation, though smart objects are also gaining adoption. All engines provide at least basic LoD support for managing NPC populations at different distances from the player.

There is also a consistent trend toward data-driven systems with visual tools that allow designers to configure NPC behaviors without requiring code modifications. This reduces the dependency on programmers for behavioral iteration, though most of these tools remain specific to their respective engines.

A common limitation across all examined engines is that their AI tools are tightly coupled to the engine's internal systems. NPC behaviors created in one engine cannot be transferred to another without substantial reimplementation. This means that studios changing engines, or developers targeting multiple platforms with different engines, must rebuild their NPC behavior systems for each engine. The few existing engine-agnostic frameworks for game AI, such as Menge [13], primarily address spatial movement and crowd navigation rather than behavioral decision-making. This gap in engine-agnostic behavioral frameworks motivates the architecture described in Chapter 4.

2.5 Commercial Video Games Case Studies

This section examines how commercial video games implement ambient NPC behavior, illustrating the practical trade-offs of the algorithms and systems described in the preceding sections. The games analyzed here were selected because they represent different approaches to ambient NPC behavior and collectively illustrate the recurring limitations identified in Section 1.2. Figure 2.6 presents screenshots from several of these titles.

2.5.1 Assassin's Creed Unity

Assassin's Creed Unity demonstrates the challenge of managing ambient NPC behavior at large scale. The game renders crowds of up to 10,000 characters but limits full AI processing to approximately 40 NPCs at a time through a layered LoD system. Characters close to the player use behavior trees with reactive responses, while distant NPCs use simplified FSMs with minimal behavior logic. The developers reported that the crowd system consumes approximately 20% of the total CPU budget [12].

Despite the scale of its crowds, the ambient NPCs in this game show repeated patterns in their animations and character models due to the limited number of behavioral variations available. The focus on rendering large crowds means that individual NPC interactions are minimal. Group formations can appear artificial because the system prioritizes rendering optimization over realistic crowd distributions. Characters also appear and disappear as they move between zones, and players experience a lack of persistence in the ambient characters.

Figure 2.6: Ambient NPCs in commercial video games. From left to right, top to bottom: *Assassin's Creed Unity*, *The Sims 4*, *Skyrim*, *Red Dead Redemption*, *Watch Dogs: Legion*, and *Cyberpunk 2077*.

2.5.2 The Sims

The Sims series features one of the most comprehensive smart object systems in commercial games, making it relevant to the entity model described in Chapter 3. In these games, objects declare what interactions they support rather than having characters contain interaction logic for every object type. This object-driven approach allows the game to be highly data-driven: designers can add new interactions by defining them on objects rather than modifying character code.

Characters in *The Sims* use a system of commodities (internal states such as hunger, energy, and social needs) combined with utility functions. Rather than evaluating every possible action, the system first determines which commodity is most important to a character and then evaluates actions that address that commodity, reducing the decision search space.

The game also uses a two-level LoD system based on distance: characters not being directly observed by the player receive simplified behavioral processing. The most recent entry in the series, *The Sims 4*, extends this with a data-driven approach where designers create NPC behaviors by combining modular components through a visual scripting system [22].

The main limitations become apparent when the player is not directly controlling characters. Autonomous behaviors become noticeably simpler, and there is minimal simulation of what characters are doing when the player is not observing them. This illustrates a recurring tension in ambient NPC design: the computational cost of maintaining complex behavior for characters the player may not be watching.

2.5.3 Skyrim

Skyrim's ambient NPC system combines FSMs with utility calculations in a hybrid approach called Radiant AI. Characters use FSMs for their basic behavioral states (working, sleeping, eating), while utility calculations determine which specific action to perform within each state. For example, when in a "working" state, utility scores based on factors like item value, character skill, and material availability determine which item a craftsman will produce. Every NPC has a primary schedule, but characters dynamically select goals based on their needs and respond to environmental changes [5].

Skyrim is notable for the limitations it reveals in schedule-based hybrid systems. Social interactions between NPCs are limited, which undermines the illusion of villages and communities. The knowledge propagation system produces unrealistic results: a single NPC witnessing a crime can cause entire settlements to become instantly hostile toward the player. Most relevant to the behavioral continuity concerns addressed in this report, *Skyrim* demonstrates incomplete interruption handling. NPCs sometimes abandon critical activities (such as fleeing from danger) to resume their scheduled behaviors, indicating that the interruption mechanism can fail to maintain appropriate behavioral continuity. The dialogue system also frequently interrupts ongoing conversations, further breaking the illusion of coherent NPC behavior.

2.5.4 Red Dead Redemption

The *Red Dead Redemption* series represents one of the most ambitious implementations of ambient NPC behavior in commercial games. In the first game, towns use a system of interaction spots that function similarly to smart objects. Each spot contains information about who can use it, at what time, and the behavior tree that governs character actions at that location. Some spots require multiple characters; for example, a poker table is only active when enough characters are present.

Red Dead Redemption 2 expands on these systems by combining schedule systems with behavior trees. Characters follow daily schedules that determine their location and general activity (e.g., a shopkeeper opens their store at 8 AM), while behavior trees handle moment-to-moment actions and responses to player interactions within each scheduled activity. NPCs have distinct personalities enriched by mood states and a memory system that is one of the most advanced in commercial games. NPCs remember the player character's appearance,

past interactions, and can recognize the player character after changes in clothing or facial hair [53].

Despite these systems, extended observation reveals that NPC routines follow strict patterns with limited variability, eventually making them feel mechanical. The thresholds for NPC reactions are inconsistently calibrated: some minor actions against an NPC receive no response while slightly more serious ones trigger disproportionate reactions. Information propagation through the memory system is also inconsistent, with some knowledge spreading very quickly between NPCs and other knowledge remaining isolated. Despite the appearance of social connections, most NPCs interact primarily with the environment or the player rather than with each other. These limitations become apparent only after extended observation, which is why this game is considered one of the most convincing implementations of ambient NPC behavior to date.

2.5.5 Watch Dogs Legion

Watch Dogs: Legion takes a procedural approach to ambient NPC behavior. Every NPC in the game is procedurally generated with consistent census data (including gender, name, occupation, and socioeconomic status) that determines their available activities and daily schedules. Activities are scheduled in 24-hour blocks subdivided into hourly segments, and the schedule for each character is generated dynamically based on their census data. Group activities are defined by character relationships. The game uses a three-level LoD system with a simple memory persistence mechanism that propagates across character relations [1, 19].

While the procedural generation creates diversity in character backgrounds, several limitations affect behavioral quality. Daily schedules contain inconsistencies, and social interactions between characters are limited. The procedurally generated background stories have almost no impact on observable NPC behavior and are primarily presented as text for the player to read. The persistence system can also produce unbelievable scenarios where an NPC maintains the same position or state for an extended period when outside the player's view.

2.5.6 Other Notable Implementations

Several other commercial games provide additional examples of the trade-offs in ambient NPC behavior.

Grand Theft Auto V (GTA V) combines schedules and behavior trees for its ambient NPCs. Different character types appear in different areas based on time of day, and pedestrians demonstrate coordinated responses to events. A spawning system manages the appearance and disappearance of ambient characters based on the player character's location, speed, and attention focus. Some NPCs maintain limited memory of hostile player character interactions. However, NPCs quickly forget dramatic encounters and return to normal behaviors, and they show repeated reactions with inconsistent recognition of environmental threats.

Cyberpunk 2077 populates its city with thousands of ambient NPCs using a schedule-based system combined with LoD techniques. NPCs closer to the player exhibit more complex behaviors and animations, while distant characters are simplified. The game includes a crowd density setting that allows players to adjust NPC populations based on their hardware capabilities. However, daily routines are limited, and characters frequently appear and disappear when the player turns away briefly or moves short distances, breaking the illusion of a persistent world.

Just Cause 3 uses a combination of behavior trees and utility functions in a decentralized approach where each NPC independently evaluates its situation. The developers created external behavior trees as modular components that can be composed into sequences of actions [36]. However, ambient characters default to generic flee behaviors in most disruption contexts, demonstrate poor environmental awareness, have limited daily routines, and retain almost no memory of player actions.

Hitman uses situation-driven behavior trees with a shared knowledge system for its NPCs. A sensor system provides characters with awareness of game events, and a services system updates shared knowledge. Characters can execute behaviors individually or join group "situations." The game uses a LoD system that updates distant NPCs less frequently, while crowd NPCs use a simple FSM with three states [16, 50]. The limited behavioral patterns and memory of crowd NPCs are partly a deliberate gameplay design choice, allowing players to plan strategies around predictable NPC behavior.

2.5.7 Case Study Observations

Despite the sophisticated systems that these games employ, several recurring limitations in ambient NPC behavior can be observed across the titles studied.

Repetitive behavior patterns. Most games struggle to produce varied behavior among ambient NPCs over extended observation. Even games with advanced hybrid systems like Red Dead Redemption 2 eventually reveal repetitive behavioral patterns. The reliance on fixed schedules, deterministic state machines, or carefully tuned utility functions means that NPCs of the same type tend to behave identically under the same conditions. Producing behavioral variety at low computational cost remains an open challenge.

Poor environmental awareness. NPCs frequently demonstrate a limited understanding of the current state of their environment, performing actions that are inappropriate for the current situation. This is particularly evident in games that rely on schedule systems, where characters may continue scheduled activities regardless of changed conditions.

Limited persistence and continuity. Most games struggle to maintain convincing persistence of characters across different play sessions or when characters move to different zones. Characters appear and disappear as spawning systems manage populations, and persistence mechanisms can produce artifacts (such as NPCs frozen in the same position for extended periods). Knowledge propagation between NPCs tends to be either too fast (entire

settlements reacting to a single witnessed event) or too slow (information remaining isolated despite proximity).

Incomplete behavioral continuity during interruptions. Several games demonstrate incomplete handling of situations where NPC behavior is interrupted or cannot continue as planned. In *Skyrim*, NPCs sometimes abandon critical responses to resume scheduled behaviors, indicating that the interruption mechanism can override more appropriate actions. In *GTA V* and *Just Cause 3*, NPCs revert to default behaviors after disruptions without preserving the context of their interrupted activity. While these games implement some form of interruption response, none of the studied titles demonstrate an integrated mechanism that preserves an NPC's interrupted context for later resumption, provides configurable fallback behaviors when a behavioral path cannot continue, and prevents cascading failures when NPC configurations produce unresolvable conditions. These concerns are addressed individually and through ad-hoc mechanisms rather than as an integrated system.

Limited social dynamics. Group interactions between ambient NPCs are rare or scripted across all studied titles. Despite some games giving the appearance of social connections (*Red Dead Redemption 2*) or group activities (*Watch Dogs: Legion*), most NPCs interact primarily with the environment or with the player rather than with each other.

These observations, combined with the algorithmic trade-offs described in Section 2.1, inform the design of the framework presented in Chapter 3. The framework specifically addresses the first four limitations: behavioral variety through memory-driven selection (Contribution 1), environmental awareness through a unified condition evaluation mechanism (Contribution 2), behavioral continuity through integrated resilience mechanisms (Contribution 3), and broad applicability through engine-agnostic design (Contribution 4). Social dynamics remain an area for future work, as discussed in Chapter 7.

2.6 NPC Behavior Research

This section reviews current research relevant to the contributions of this report. Section 2.6.1 examines research that directly addresses the behavioral variety and scalability trade-off that motivates our framework. Section 2.6.2 reviews memory representation research, which is relevant to our framework's memory-driven selection mechanism. Section 2.6.3 briefly surveys other research directions that, while not directly addressed by our framework, represent potential extensions or complementary approaches.

2.6.1 Behavioral Variety and Scalability

The trade-off between behavioral variety and computational scalability identified in Section 1.2 has been addressed by prior work through strategies that differ fundamentally from the approach proposed in this report.

Bulitko et al. [7] directly target behavioral variety for ambient NPCs by proposing an evolutionary approach that generates large sets of NPC behaviors offline and trains deep

neural networks to identify the most interesting behaviors. Their work addresses the problem of repetitive behavior through offline evolutionary computation and learned classifiers that generate behavior policies before gameplay begins. This approach produces varied behaviors but requires a separate training phase before deployment. In contrast, our framework produces behavioral variety at runtime through a lightweight memory-driven selection mechanism that requires no offline training, evolutionary search, or learned models.

As discussed in Section 2.1.3, Cardon and Jacopin [9] address scalability from a different angle by parallelizing GOAP planning on a GPU, enabling planning for thousands of NPCs per frame. Their approach retains the behavioral variety that planning algorithms provide but shifts the computational cost to specialized hardware rather than reducing it algorithmically. Our framework instead replaces per-decision planning with a bounded memory mechanism that achieves variety at a cost comparable to reactive methods, without requiring GPU resources that may compete with rendering.

The case studies in Section 2.5 illustrate a third strategy used in commercial games: managing cost through complexity reduction and temporal distribution rather than modifying the decision-making algorithm itself. *Assassin's Creed Unity* limits full AI processing to approximately 40 NPCs through LoD scaling [12], while *Hitman* distributes AI updates across frames using time-slicing [16, 50]. These techniques manage cost effectively but do not address behavioral variety: NPCs at lower detail levels or with less frequent updates still follow the same deterministic behavioral patterns.

Our framework addresses the variety-scalability trade-off at the algorithmic level. Rather than accelerating expensive decision-making (Cardon and Jacopin), generating varied behaviors offline (Bulitko et al.), or reducing the number of NPCs receiving full behavioral processing (commercial LoD approaches), our framework replaces per-decision optimization with a bounded memory system that produces varied behavior at low runtime cost.

2.6.2 Memory Representation

Memory representation research focuses on modeling the knowledge and experiences of NPCs to enable persistent and adaptive behaviors that respond to past events. This research is relevant to our framework because our memory-driven selection algorithm (Contribution 1) uses a form of NPC memory, though for a different purpose than most systems described in the literature.

Recent research has explored several approaches to NPC memory representation. Episodic databases store records of events including actions, locations, and outcomes. Chronological timelines organize events by time sequence, enabling NPCs to recall what happened and when. Associative networks link related experiences together, allowing retrieval of memories based on contextual similarity rather than strict temporal ordering. Supporting mechanisms such as retrieval systems (for accessing relevant memories efficiently), forgetting mechanisms, and interaction tracking (for recording encounters with the player and other NPCs) have also been explored [43, 55].

Park et al. [43] demonstrated one of the most comprehensive NPC memory systems in their Generative Agents research, where characters maintain detailed memory streams of their experiences and use a large language model to retrieve and reflect on relevant memories

when making decisions. Their system produces highly varied and contextually rich behavior but requires substantial computational resources (including LLM inference for each decision), making it impractical for the large ambient NPC populations that our framework targets.

Zheng et al. [55] proposed MemoryRepository, a structured memory system for NPCs that organizes memories into categorized repositories with retrieval and forgetting mechanisms. Their work focuses on enabling NPCs to maintain coherent long-term behavior based on accumulated experiences.

The memory system in our framework differs from these approaches in both purpose and scope. Existing memory research primarily models what an NPC has experienced (perceptual and episodic knowledge) to produce contextually appropriate responses to past events. Our framework's memory system instead tracks what decisions an NPC has made recently (transition choices and entity selections) to guide future choices away from repetition. This is a narrower scope that enables a simpler data structure (bounded lists of recent decisions rather than rich episodic records) with lower per-character memory requirements and computational cost. The bounded capacity of our memory system also introduces natural forgetting that prevents predictable behavioral patterns, as described in Chapter 3. While our framework's memory system does not model the rich experiential knowledge that systems like Generative Agents provide, this simplicity is what enables it to scale to hundreds of ambient NPCs within the frame budget constraints described in Section 2.3.3.

2.6.3 Other Research Directions

Several other research areas are relevant to ambient NPC behavior, though they address concerns outside the scope of the contributions proposed in this report. These areas represent potential extensions or complementary approaches that could be combined with our framework in future work.

Emotion models create computational representations of emotional states that influence NPC decision-making. Research has explored cognitive models such as the OCC model (which categorizes emotions based on evaluations of events, actions, and objects) and the Pleasure-Arousal-Dominance model (which represents emotional states in a three-dimensional space). Integrating personality traits with emotion models can create more distinctive character behaviors [3, 4, 31]. For ambient NPCs, simplified emotional models could enrich behavioral variety by influencing action selection, though the high computational cost of accurate emotional modeling for large populations remains a challenge.

Social simulations model relationships, social norms, and group behaviors that determine interactions between NPCs. Research has focused on graph-based social networks where characters are nodes and relationships are weighted edges, as well as on group dynamics including formation, dissolution, and coordination of collaborative activities [10, 33, 34]. Ambient NPCs could benefit from simplified relationship models that influence group behaviors, such as characters who share a "works with" relationship coordinating their activities. Our framework's current design does not model social relationships, but its entity model and memory system could be extended with additional memory types to support social interactions, as discussed in Chapter 7.

Machine learning approaches use data-driven methods to create or enhance NPC

behaviors. Research has explored imitation learning (where NPCs learn by observing human players), reinforcement learning, and generative models that produce context-appropriate behaviors [2, 38, 39]. The main limitations for ambient NPC applications are the runtime cost of model inference and the difficulty of obtaining appropriate training data for ambient character behaviors. These approaches could complement our framework by, for example, using learned models to generate behavioral content (action and sequence definitions) offline rather than during gameplay.

Generative AI approaches use large language models or other generative techniques to create dynamic NPC behaviors, including interpreting game state to suggest appropriate actions, generating contextually appropriate dialogue, and creating narrative-appropriate action sequences [8, 14, 25, 32, 40, 49, 54]. The primary limitations are the computational requirements for runtime inference and the challenge of balancing generative creativity with the behavioral consistency that games require. For ambient NPCs, the most practical applications may be in offline content generation (producing behavioral specifications) rather than runtime decision-making.

3. Framework Theoretical Model

Our analysis of existing algorithms for NPC behaviors in Chapter 2 reveals a fundamental trade-off in ambient NPC behavior systems. Algorithms that produce varied, context-aware action selection are too computationally expensive for the large ambient NPC populations that open-world games require, while algorithms that scale well produce repetitive patterns that players notice. The most comprehensive existing framework specifically addressing ambient NPCs (CIVIL AI, described in Section 2.4.1) integrates multiple algorithms, but is tightly coupled to a single game engine, limiting its broader applicability.

Our framework addresses this trade-off through a different strategy. Rather than calculating optimal choices for each NPC's action selection, our approach remembers what each NPC has done recently and uses that history to guide future choices. This keeps the per-decision computational cost low enough for handling large NPC populations while producing the behavioral diversity typically associated with more computationally expensive algorithms. Our framework is engine-agnostic and data-driven: NPC behaviors are defined through configurable data rather than hard-coded logic, allowing developers to define and modify behaviors without changes to the framework code. This chapter presents our framework at a conceptual level.

3.1 Framework Components

3.1.1 Entities

In many existing NPC behavior systems, the logic for interacting with objects is embedded in character behavior code. An NPC contains specific logic for each type of object it encounters (e.g., a storage crate or a chair for resting). As the number of object types increases, this approach becomes difficult to maintain because adding a new object type may require modifying the behavior code of every character that can interact with it. An alternative explored in commercial games such as The Sims (Section 2.5.2) is the concept of smart objects, where objects declare what interactions they support rather than relying on characters to contain that knowledge (Section 2.2.1).

Our framework adopts and extends this principle through an entity model in which all interactive elements in a game (e.g., benches, doors, food stalls, and the characters themselves) are represented as **entities**. Rather than maintaining separate representations for objects and characters, our framework uses a single structure for all elements involved in NPC behavior. Each entity maintains two types of information: **state** and **supported actions**.

State: The properties that describe the current condition of an entity. A park bench might have properties representing how many of its seats are currently occupied; a food stall might track its current stock level and whether it is open for business; an NPC might track its energy level, current mood, and whether it is sitting. These properties change over time

as a game progresses.

Supported Actions: Each entity declares which actions it supports, representing what can be done with or on the entity. A park bench might support sitting, a food stall might support buying food and browsing its goods, and an NPC might support socializing. Supported actions allow our framework to identify which entities in a game are relevant to an action that an NPC is going to perform. The formal definition of actions and how they interact with entities is presented in Section 3.1.3.

Within this entity model, our framework distinguishes between two roles: **framework entities** and **behavioral entities**.

Framework entities: These are the base interactive elements of a game (e.g., the benches, doors, stalls, and other objects that NPCs can interact with). These entities maintain their state and declare their supported actions, but they do not make decisions about what to do next.

Behavioral entities: These are the NPCs of a game. Just like any framework entity, behavioral entities have state and action declarations. Additionally behavioral entities possess decision-making systems that enable our framework to control their behavior.

This distinction enables a consistent interaction model. When an NPC sits on a bench, both the NPC and the bench are entities whose properties can be examined and modified through the same mechanism (described in the next section). The same applies when two NPCs interact with each other (e.g., when socializing): Both participants are entities with compatible properties and supported actions.

Table 3.1 shows example entities in a park scenario. The two benches and the food stall are **framework entities**: They participate in NPC actions by maintaining state and declaring supported actions, but they do not act on their own. The visitor NPC is a **behavioral entity** with decision-making systems that determine which bench to sit on or whether to browse the food stall. The components through which these decisions are made are described in the remaining sections of this chapter.

Entity	Properties (State)	Supported Actions
Park Bench A	occupied seats: 1/3	sit on bench
Park Bench B	occupied seats: 3/3	sit on bench
Food Stall	stock level: 12; open status: open	buy food; browse stall
Visitor NPC	energy: 65; mood: happy; sitting: false	socialize

Table 3.1: Entities with state and supported actions in a park scenario.

3.1.2 State Operations

Throughout the decision-making processes described in this chapter, our framework checks the current state of entities and makes appropriate state changes. For example, before an

NPC sits on a bench, our framework checks whether the bench has available capacity. As the NPC sits, our framework increases the bench's occupied seat count. These two types of operations, evaluating conditions and modifying state, appear throughout our framework in different contexts: Determining whether an NPC can perform a particular action, deciding which action to perform next based on current game conditions, and updating entity states after an interaction occurs.

A naive approach would implement separate evaluation mechanisms for each of these contexts, with dedicated logic for entity properties, behavioral properties, and game environmental conditions (e.g., weather or time of day). This would create inconsistencies and couple the framework to the internal data structures of a specific game. Instead, our framework uses a single mechanism for all condition evaluations and state modifications. Whether the framework checks an NPC's energy level, a bench's available capacity, or the current weather, it uses the same approach. This allows NPC behaviors to be defined independently of how characters, objects, and environmental conditions are represented within a game.

This mechanism, which we call **state operations**, is defined by four properties. The **target** specifies what is examined or modified: The NPC itself, another entity (e.g., a bench), a game environmental condition (e.g., weather or time of day), or a derived value such as the distance to another entity. The **state property** identifies a specific attribute of the target, such as a bench's "occupied-seats" or an NPC's "energy level." The **operation type** determines how the property is used, either as a comparison (e.g., checking whether a value meets a threshold) or a modification (e.g., incrementing or setting a value). The **operation value** specifies the threshold for comparisons or the amount to apply for modifications.

Table 3.2 shows examples of state operations from the park scenario. The first three rows evaluate conditions for decision-making, while the last two modify states to reflect the outcomes of actions. The same four-property structure applies regardless of whether the target is an NPC, a framework entity, or a game environmental condition.

Target	**State Accessed and Operation**	**Result / Update**
Park Bench A	occupied seats (1) <= 3	true
Food Stall	open status == open	true
Visitor NPC	energy (65) <= 70	true
Park Bench A	occupied seats += 1	from 1 to 2
Visitor NPC	set sitting = true	from false to true

Table 3.2: State operations in the park scenario.

Game **environmental conditions** require special consideration. Properties like weather, time of day, or location characteristics are managed by game engine systems rather than by entities within our framework. When an NPC evaluates a state operation targeting a game environmental condition, our framework must obtain that value from game engine systems. If every NPC individually queried the game for this information on each decision, the resulting volume of queries could become a performance concern in games with hundreds of ambient characters.

To address this, our framework maintains a cache of game environmental condition values. The framework periodically requests updated values from game engine systems and stores them so that all NPCs access cached values during state operation evaluations. Each cached condition has a configurable refresh interval: Conditions that change rarely (e.g., the current season) can be refreshed infrequently, while conditions that change often (e.g., time of day) can be refreshed more frequently. This approach ensures that the cost of environmental awareness is not multiplied by the number of characters.

3.1.3 Actions

Actions are the fundamental building blocks of NPC behavior in our framework. Each action represents an individual behavior that an NPC can perform, such as sitting on a bench, buying food from a stall, or walking to a location. An NPC sitting on a bench is a single action; a routine involving resting, buying food, and walking around is a larger behavioral pattern composed of multiple actions. The way actions are organized into these larger patterns is described in Section 3.1.4.

Our framework is responsible for deciding *when* an action should start and for managing the state changes associated with it. A game engine is responsible for *how* an action is executed visually and audibly, including animations, sounds, and character movement. When our framework determines that an NPC should sit on a bench, it signals game engine systems to begin the corresponding visual execution. Game engine systems handle the specifics (e.g., how the NPC walks to the bench and sits down) and signal our framework when the action is complete. This separation allows the same action definition to produce different visual outcomes depending on the game engine or available visual and audio resources, without requiring changes to the behavioral logic.

Action Properties

Each action in our framework is defined by the following properties:

Preconditions: State operations that must all evaluate to true before an NPC can perform an action. Preconditions are the primary mechanism through which our framework achieves context-aware behavior at the action level. For example, a "sit on bench" action might require that the target bench has fewer occupied seats than its total capacity and that the current weather is not "rainy." If the bench is full or it is raining, the NPC will not attempt to sit. This means that the same NPC can behave differently under different game conditions without requiring separate behavioral logic for each situation.

Effects: Effects are state operations that modify entity states at specific moments during an action's execution. Our framework defines three types of effects, each applied at a different stage. **Immediate effects** are applied when an action begins. These effects typically reserve resources or update states to reflect that an action is underway. For the "sit on bench" action, immediate effects might increase the bench's "occupied-seats" count by one and set

the NPC's "sitting" state to "true." These changes happen at the start so that other NPCs evaluating the same bench see accurate state information.

Completion effects are applied when an action finishes. These represent the outcomes of having performed the action. For the "sit on bench" example, a completion effect might increase an NPC's "energy" by a certain amount, reflecting the rest gained from sitting.

Interruption effects are applied if an action is suspended before it completes. When game events require an NPC to stop its current action and respond to something more urgent (the full mechanism is described in Section 3.3.4), interruption effects maintain state consistency. For the "sit on bench" action, interruption effects would decrease the bench's "occupied-seats" count and set the NPC's "sitting" sta te back to false. Without these effects, a bench could permanently lose a seat every time an NPC is interrupted while sitting.

Resumability: This property indicates whether an action that has been interrupted can be continued from where it stopped or must start over. For example, an NPC sitting on a bench when interrupted might return to the same bench afterward (resumable), while an NPC mid-greeting gesture would restart the greeting from the beginning (non-resumable). The mechanisms for handling interruptions and resumptions are described in Section 3.3.4 and Section 3.3.5.

Duration: This property defines how long an action is expected to take under normal circumstances. This value is communicated to a game engine when an action begins, allowing it to manage the execution appropriately (e.g., determining the length of a sitting animation and rest period). The game engine signals our framework when the action's execution is complete.

Maximum timeout: A safeguard that defines a time limit after which our framework considers an action complete, regardless of whether a completion signal was received from a game engine. If a game engine fails to signal completion due to a bug or communication issue, this timeout ensures that NPCs continue making decisions rather than becoming stuck indefinitely. This value is longer than the expected duration, providing a margin for normal variation in execution time.

Action categories

Actions fall into two categories based on whether another entity is needed to perform them. **Entity-requiring actions** need a specific entity to interact with. "Sit on bench" requires a bench, "buy food" requires a food stall. When our framework determines that an NPC should perform an entity-requiring action, it must identify and select an appropriate entity from those available in a game. The process for how entities are identified, filtered by preconditions, and selected is described in Section 3.3.3.

Independently-performed actions can be performed by an NPC without interacting with another entity. Actions like stretching, looking around, or resting in place only involve the NPC itself. For these actions, the state operations in preconditions and effects target only an NPC's own state or game environmental conditions.

3.1.4 Behavior Sequences

Individual actions alone do not create interesting NPC behavior. An NPC that can sit, buy food, and walk around, has the building blocks for varied activity, but without a way to organize these actions into meaningful patterns, it has no way to determine what to do, in what order, or under what conditions. **Behavior sequences** provide this organization. A behavior sequence is a behavioral pattern that connects actions through conditional decisions, defining the possible behaviors that an NPC can exhibit. Rather than encoding a fixed series of actions, a behavior sequence captures both the structure and the conditions that determine how an NPC's decision-making systems determine what the NPC will do.

Directed Graph Structure

Formally, a behavior sequence is a directed graph where the actions are nodes and the transitions are directed edges. Each behavior sequence designates one node as its entry point, which is where execution begins when an NPC starts following a sequence. The directed edges (transitions) define the possible paths through the graph, and preconditions on those edges determine which paths are available under current conditions.

Figure 3.1 illustrates this structure using a "park visitor" behavior sequence. The behavior sequence begins at a "rest" action node as the entry point. From this node, transitions lead to a "sit on bench" action node or to a "buy food" nested sequence node. Both of these nodes transition to a "walk around" action node, which itself transitions to the end sequence node. This structure allows the same behavior sequence to produce different behavioral outcomes depending on current game conditions. An NPC whose "energy" is 80 would take a different path than when its "energy" is 50.

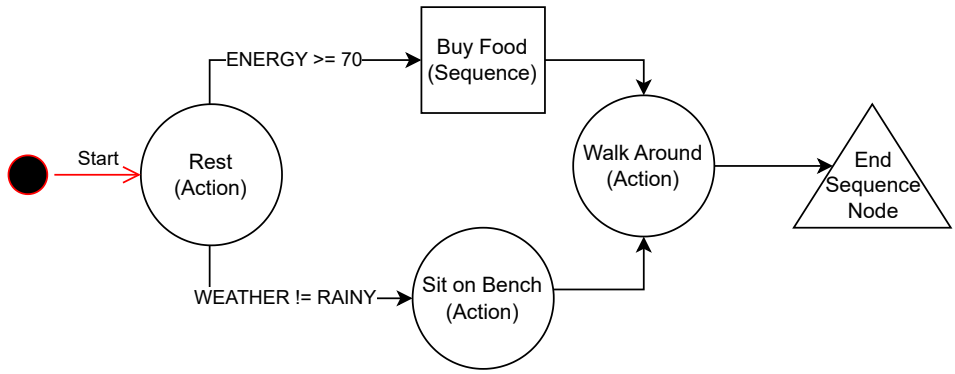

Figure 3.1: Example of a behavior sequence.

Sequence Nodes

Behavior sequences are composed of nodes, where each node represents an action within a behavioral pattern. Our framework defines three types of nodes:

Action Nodes reference an action (as defined in Section 3.1.3) that an NPC will attempt to perform when this node is reached. The process for initiating actions, including precondition evaluation and entity selection, is described in Section 3.3.3.

Nested Sequence Nodes reference a behavior sequence rather than an individual action, allowing complex behaviors to be composed from simpler, reusable patterns. For example, the "park visitor" behavior sequence might contain a nested sequence node referencing a "buy food" behavior sequence, which itself contains action nodes for walking to a food stall, browsing, and purchasing food. This "buy food" behavior sequence is defined once and can be referenced across multiple sequences. When a nested sequence node is reached, the referenced behavior sequence executes, and upon its completion, the original behavior sequence resumes. The mechanism that manages this nesting is described later in this section under "Sequence Stack."

End Nodes mark the completion of a behavior sequence. When an end node is reached, the current behavior sequence is considered finished. What happens next depends on whether the behavior sequence was invoked by another sequence (in which case execution returns to the calling behavior sequence) or whether it is the NPC's primary behavioral pattern (in which case the pattern restarts, as described under "Main Sequence" below).

Transitions

Transitions define the connections between nodes in a behavior sequence. Each transition leads from one node to another and may optionally include preconditions (defined as state operations) that must evaluate to true for the transition to be available. While action preconditions (Section 3.1.3) determine whether a specific action can be performed, transition preconditions determine which paths through a behavior sequence are available, allowing an NPC's progression to adapt to current game conditions.

A single node may have multiple outgoing transitions, each with its own preconditions, creating decision points where different paths become feasible depending on game conditions. For example, from the "rest" node in Figure 3.1, one transition leads to "buy food" with the precondition that the NPC's energy is at least 70, while another leads to "sit on bench" with the precondition that the weather is not "rainy." If both preconditions are satisfied, both transitions are feasible and the NPC must choose between them. Transitions may also have no preconditions, in which case they are always feasible once the current node's processing has completed. The mechanism by which an NPC selects among multiple feasible transitions is described in Section 3.1.5.

Sequence Stack

Every behavioral entity is assigned a **main behavior sequence**, which defines its default behavior over time (e.g., a vendor's daily routine, a park visitor's activities, or a guard's patrol route). Since behavior sequences can contain nested sequence nodes, each NPC requires a mechanism to track which behavior sequence is currently executing. Our framework provides this through a **per-NPC stack of active sequences**. Figure 3.2 shows how the sequence stack is used: (1) When an entity is initialized, its main behavior sequence is pushed onto its sequence stack as the starting point for execution. (2) When a nested

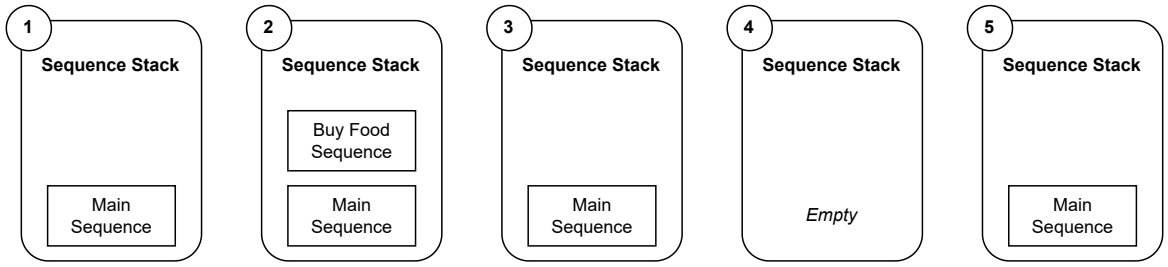

Figure 3.2: Example of progression of a sequence stack.

sequence node is reached (e.g. the buy food sequence node from Figure 3.1), the referenced behavior sequence is pushed onto the top of the stack and becomes the new active sequence. (3) When the nested sequence reaches an end node, it is removed from the stack, and the previous behavior sequence resumes from the point where the nested sequence was initiated. (4) If the stack becomes empty (i.e., all active sequences have reached their end nodes and been removed), (5) the framework automatically pushes the main behavior sequence back onto the stack. This ensures that an NPC always has an active behavior sequence to follow.

3.1.5 Memory System

The components described so far provide the structural foundation for NPC behavior but do not produce behavioral variety on their own. Consider two NPCs in the park scenario, both following identical "park visitor" behavior sequences. When both reach a decision point with transitions to "sit on bench" and "browse food stall", both evaluate the same preconditions against the same game conditions. If both transitions are valid, nothing in the mechanisms described so far distinguishes one choice from another. The same problem applies to a single NPC restarting its behavior sequence: Given unchanged game conditions, it would follow the same path every time. Our framework addresses this through a memory system: Rather than calculating which action is best (as utility systems and planning algorithms do at higher computational cost, described in Section 2.1), the framework tracks what each NPC has done recently and uses that history to guide future choices. This keeps the per-decision cost low while producing behavioral diversity.

Memory Record Types

Each NPC maintains a collection of memories that record its recent decisions. Our framework defines three types of **memory records**, each tracking a different kind of decision that NPCs make when following sequences.

Transition memories record which paths an NPC has recently chosen at each decision point within a behavior sequence. Each transition memory references both the selected transition and the behavior sequence in which the selection occurred, because different sequences might contain common nodes and the memory system must distinguish between choices made in different behavioral contexts. For example, if the visitor NPC reaches a decision point and follows the transition to "sit on bench" rather than "browse food stall,"

a transition memory records this selection within the current behavior sequence. The next time the NPC reaches the same decision point, this memory influences which path is chosen.

Action memories record which entity an NPC selected when performing an entity-requiring action. Each action memory references both the selected entity and the action that was performed, because the same entity might support multiple actions and the memory system must distinguish between using a bench for sitting versus using it for reading. For example, if the visitor NPC sits on "Park Bench A," an action memory records that "Park Bench A" was used for the "sit on bench" action. The next time the NPC needs to sit, this memory influences which bench is selected, encouraging the NPC to try "Park Bench B" instead.

Interruption memories serve a different purpose from the other two types. Rather than creating variety, they preserve the context needed to resume an action that was interrupted before completion. When a resumable action is interrupted, an interruption memory stores references to the action being performed, the behavior sequence it belongs to, the node within that sequence, and the entity being used (if applicable). This allows our framework to attempt continuing the interrupted action after the interruption concludes, rather than forcing the NPC to restart its behavior sequence from the beginning. Once the context is used for a resumption attempt, the interruption memory is removed regardless of whether the attempt succeeds. The full mechanisms for interruption and resumption are described in Section 3.3.4 and Section 3.3.5.

All three memory types also record when the memories were created. This timestamp is essential for the selection algorithm later in this section, which uses recency to determine how memories influence future decisions.

Memory Management

The memory system follows a consistent process for creating and updating all three memory types. When a new memory is created (after a transition is selected, an entity is chosen for an action, or a resumable action is interrupted), the system first checks whether a memory for the same decision already exists. For transition memories, "the same decision" means a memory referencing the same destination node in the same behavior sequence; for action memories, the same entity for the same action; for interruption memories, the same action at the same node in the same behavior sequence. If a matching memory exists, it is removed before the new one is added, ensuring that only the most recent occurrence of a decision is tracked. The new memory is then added with the current time as its creation timestamp.

Capacity and Forgetting

Each NPC has configurable capacity limits for each memory type, defining how many memories of each type can be retained at once. When adding a new memory causes a type to exceed its capacity, the oldest memory of that type is removed. This bounded capacity serves two purposes.

First, it controls the memory footprint of each NPC. In a game with hundreds of ambient characters, unbounded memory growth would create a scalability problem. Configurable

limits allow developers to balance behavioral variety against resource usage.

Second, bounded capacity introduces natural forgetting that prevents perfectly predictable behavioral patterns. Consider an NPC with a transition memory capacity of five, navigating a behavior sequence with eight possible paths. After the NPC has explored five different paths, the earliest memories are removed as new ones are added, so paths explored long ago become available for selection again. This creates a natural rhythm where NPCs cycle through available options without following a rigid rotation that players could predict.

Interruption Memory Cleanup

Interruption memories have additional removal conditions beyond capacity-based. As described above, an interruption memory is removed after it is used for a resumption attempt. Additionally, when a behavior sequence encounters conditions that prevent it from continuing (described in Section 3.3.6), all interruption memories associated with that sequence are removed. This prevents preserved context from a sequence that can no longer proceed from interfering with future executions of the same sequence.

Memory-Driven Selection Algorithm

When an NPC faces a choice between multiple valid options (either selecting among transitions at a decision point or choosing among entities for an action), our framework applies a selection algorithm that uses the NPC's memories to guide the decision. The algorithm receives two inputs: The set of valid options and the relevant memory records for the current decision type (transition memories when selecting transitions, action memories when selecting entities). The algorithm proceeds in three steps:

1. **Prefer options not recently selected.** If any valid options are not represented in the NPC's current memory records (i.e., there is no memory record indicating that the option was selected recently), randomly select among these options. This encourages behavioral variety by favoring options that have not been selected in the recent past.

2. **Otherwise, select the least recently chosen option.** If all valid options are represented in memory (i.e., each option has an associated memory record), select the option whose memory record has the oldest creation timestamp. This distributes selections across all options over time and prevents frequent repetition of the same choice.

3. **Break ties randomly.** If multiple options are equally eligible (e.g., multiple options without memory records in Step 1, or multiple options with equally old memory timestamps in Step 2), select randomly among them. This avoids deterministic patterns and ensures variation across NPCs with otherwise identical configurations.

Figure 3.3 shows a flowchart of this algorithm. To illustrate, consider an NPC at a decision point with three valid transitions, leading to nodes A, B, and C. The NPC's transition memories show that A was selected 30 seconds ago and B was selected 60 seconds ago, but no memory exists for C. The algorithm selects C (Step 1: an unrecorded option). Suppose

the NPC later returns to this decision point, and now memories exist for all three nodes: A (40 seconds ago), B (70 seconds ago), and C (10 seconds ago). The algorithm selects B (Step 2: the oldest memory). If instead both A and B had been selected at exactly the same time, the algorithm would randomly choose between them (Step 3). Multiple options with identical timestamps should not occur in practice, but Step 3 handles this edge case.

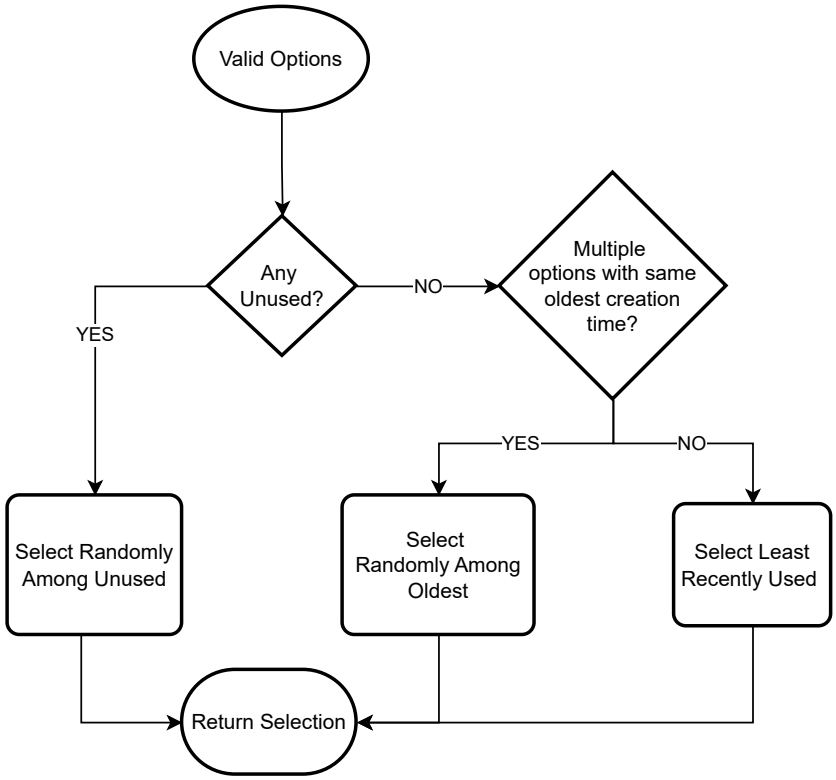

Figure 3.3: Memory-Driven Selection Algorithm Flowchart.

This algorithm produces emergent behavioral variety from simple operations. Different NPCs following identical behavior sequences diverge in their choices because their individual memory histories differ after the first few decisions, and the same NPC makes different choices across repeated executions as its memories evolve. The per-decision cost is limited to checking a small number of recent memory records against the available options.

3.2 Conceptual Architecture

The previous section introduced the individual components of our framework. This section describes how these components are organized into groups and how those groups coordinate to produce NPC behavior.

3.2.1 Component Groups and Coordination

Our framework organizes its components into three groups based on their role in the decision-making process. The **Entity Group** encompasses the entity model described in Section 3.1.1, managing all interactive elements through the unified entity representation. The **Behavioral Group** encompasses state operations (Section 3.1.2), actions (Section 3.1.3), and behavior sequences (Section 3.1.4), defining *what* NPCs can do and *how* their behavior is organized. The **Memory Group** encompasses the memory system described in Section 3.1.5, tracking NPC decision history and providing the selection mechanism that creates behavioral variety.

These three groups maintain separate responsibilities but work together through specific interaction patterns during NPC decision-making. Figure 3.4 illustrates these patterns, showing how the Behavioral Group coordinates with the Entity Group and Memory Group while also interacting with game engine systems.

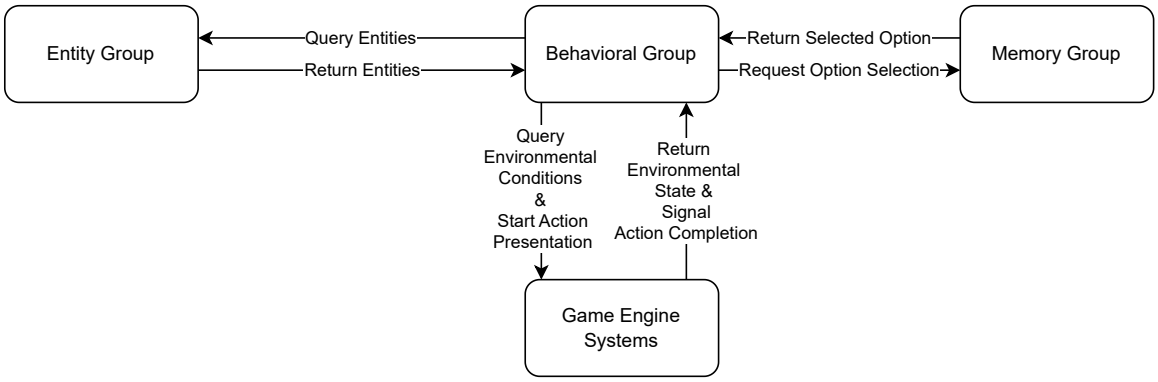

Figure 3.4: Framework Component Groups Coordination.

Entity-Behavioral coordination occurs when the Behavioral Group needs to identify entities an NPC can interact with. When an NPC reaches an action node for an entity-requiring action, the Behavioral Group queries the Entity Group for all entities that declare support for that action. The Entity Group returns these entities along with their current state, which the Behavioral Group uses to evaluate the action's preconditions and determine which entities are valid targets.

Memory-Behavioral coordination occurs when the Behavioral Group has identified multiple valid options and needs to choose among them. This happens at two points: When multiple transitions from the current node satisfy their preconditions, and when multiple entities are valid for an entity-requiring action. In both cases, the Behavioral Group delegates the choice to the Memory Group, which applies the selection algorithm (Section 3.1.5) using the NPC's relevant memories and returns the selected option.

Behavioral-Game Engine Systems coordination occurs in two situations. First, when the Behavioral Group needs to evaluate an environmental condition whose cached value has expired (Section 3.1.2), it requests an updated value from game engine systems. Second, when an action is ready for execution, the Behavioral Group signals game engine

systems to begin the action's visual and auditory execution (e.g., animation, movement, and sound). Game engine systems signal back when the action is complete.

3.2.2 Control Flow

The coordination patterns above form a continuous decision-making cycle that each NPC follows. An NPC processes its active behavior sequence step by step: It examines the current node, processes it (executing an action, entering a nested sequence, or completing the sequence), evaluates available transitions using preconditions, and selects the next node using the memory-driven selection algorithm. This cycle repeats, producing ongoing behavior.

Three situations can arise during this cycle. The first is **normal progression**, where an NPC follows its behavior sequence, performs actions, and selects transitions. The second is **interruption and resumption**, where a game event temporarily interrupts an NPC's current behavior and triggers a response; after the response completes, the NPC attempts to continue where it was interrupted. The third is **recovery from conditions preventing continuation**, where no valid path exists from the current node (e.g., no transitions have satisfied preconditions, or no entities are available for an entity-requiring action). In our framework, these situations are called **behavior failures**, though they are not software errors or crashes; they arise naturally as game conditions change. Our framework has specific recovery mechanisms to maintain NPC activity in each of these situations. The processes are described in Section 3.3.

3.3 Decision-Making Process

This section describes how NPCs execute behavior sequences within our framework. The components introduced in Section 3.1 and their coordination described in Section 3.2 define what NPCs operate on; this section focuses on the dynamic processes: How NPCs progress through behavior sequences, execute actions, respond to interruptions, and recover when conditions prevent further progression.

To ground these processes in a concrete scenario, consider the "visitor NPC" following the "park visitor" behavior sequence from Section 3.1.4, reproduced in Figure 3.5. The NPC begins at a resting action, transitions to either sitting on a bench or buying food depending on game conditions and its memory, and eventually reaches an end node.

3.3.1 Sequence Lifecycle States

Each sequence tracks its progress through a **lifecycle state**, which indicates the kind of processing the sequence requires at any given moment. Understanding these states is essential for following the execution processes described in subsequent sections. Our framework defines seven lifecycle states:

Uninitialized: The behavior sequence has been assigned to an NPC but has not yet begun execution. This is the initial state when a behavior sequence is first added to an NPC's sequence stack.

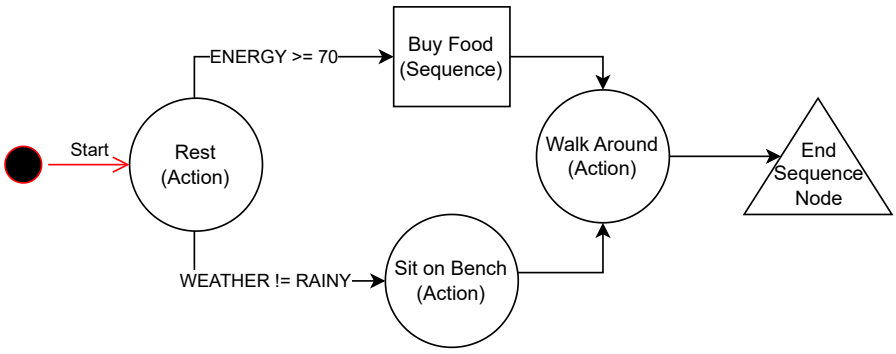

Figure 3.5: Reproduced "park visitor" behavior sequence

Processing Node: The framework is examining the current node to determine what kind of processing is required. The next state depends on the node type (action, nested sequence, or end node).

Waiting for Action: The behavior sequence is paused while the current action executes. The sequence remains in this state until the action completes.

In Subsequence: Execution of the current behavior sequence is paused while a nested sequence runs.

Node Executed: The current action or nested sequence has finished. The framework evaluates outgoing transitions to determine the next node.

Interrupted: A game event has paused the current behavior sequence while a higher-priority response sequence executes. The behavior sequence preserves its position for potential resumption.

Failed: The behavior sequence cannot continue from its current position. The framework activates recovery mechanisms (Section 3.3.6) to restore NPC activity.

3.3.2 Sequence Execution Process

This section describes the normal execution flow, following the visitor NPC through its "park visitor" behavior sequence. Figure 3.6 shows the flow diagram being described.

When a behavior sequence first becomes the active sequence on an NPC's stack, it is in the **uninitialized** state. The framework identifies the sequence's entry node, sets it as the current node, and transitions the sequence to the **processing node** state. For the visitor NPC, the "park visitor" behavior sequence begins at the "rest" node shown in Figure 3.5.

Node Processing

In the **processing node** state, the framework examines the type of the current node and acts accordingly.

If the current node is an **action node**, the framework initiates the action execution process described in Section 3.3.3. The sequence transitions to the **waiting for action**

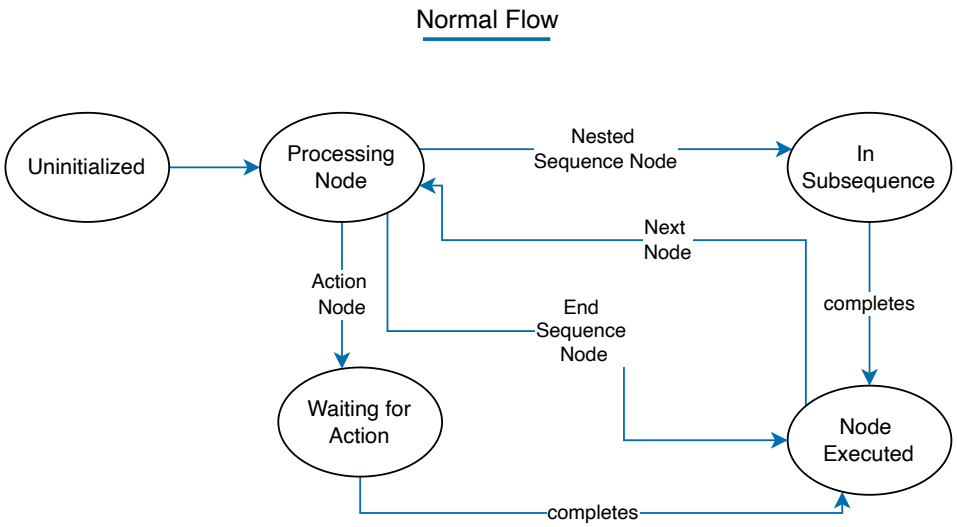

Figure 3.6: Normal Sequence Execution Flow.

state and remains paused until the action completes. For the visitor NPC, the "rest" node is an action node, so the framework begins executing the "rest" action.

If the current node is a **nested sequence node**, the framework pushes the referenced behavior sequence onto the NPC's sequence stack. The current sequence transitions to the **in subsequence** state, and the newly pushed sequence begins its own lifecycle from the **uninitialized** state.

If the current node is an **end node**, the sequence is complete and the framework removes it from the stack. If another sequence exists below it on the stack (because this was a nested sequence or an interruption response), that sequence resumes. If the stack is empty, the NPC's main behavior sequence is automatically restarted as described in Section 3.1.4.

Transition Evaluation

After an action or nested sequence completes (so the sequence is in the **node executed** state), the framework evaluates the outgoing transitions from the completed node. Each transition's preconditions are evaluated using state operations (Section 3.1.2), and only transitions whose preconditions are all satisfied are considered valid.

For the visitor NPC, suppose the "rest" action has completed. Figure 3.5 shows two outgoing transitions from the "rest" node: One leading to "sit on bench" with preconditions on energy level and weather, and another leading to "buy food" with a precondition that the food stall is open. The framework evaluates each transition's preconditions and determines which are valid given the current game conditions.

When multiple transitions are valid, the framework applies the memory-driven selection algorithm from Section 3.1.5. For the visitor NPC, if both transitions are valid, the framework checks the NPC's transition memories for the "park visitor" behavior sequence. If no memory exists for the "sit on bench" destination but a memory exists for "buy food," the

algorithm selects "sit on bench" and creates a new transition memory recording the selection. When only one transition is valid, the framework follows it directly without consulting the memory system.

The selected destination node becomes the new current node and the sequence transitions back to the **processing node** state. This cycle of processing, evaluating transitions, and selecting the next node repeats, producing the ongoing behavioral pattern that players observe.

If no outgoing transitions from the current node have their preconditions satisfied, the sequence cannot continue. The framework puts the sequence in the **failed** state and activates the recovery mechanisms described in Section 3.3.6. For example, if both transitions from the visitor NPC's "rest" node require conditions that are no longer true (the weather has turned rainy and the food stall has closed), the framework transitions to an alternative behavior sequence rather than leaving the NPC idle.

3.3.3 Action Execution Process

This section describes what happens when the framework encounters an action node during sequence execution. The process differs depending on whether the action requires interaction with another entity and whether it was previously interrupted. Figure 3.7 illustrates this flow as a swimlane diagram, where the top row shows steps performed by the framework and the bottom row shows steps delegated to the game engine. Arrows crossing between rows indicate communication at the systems boundary.

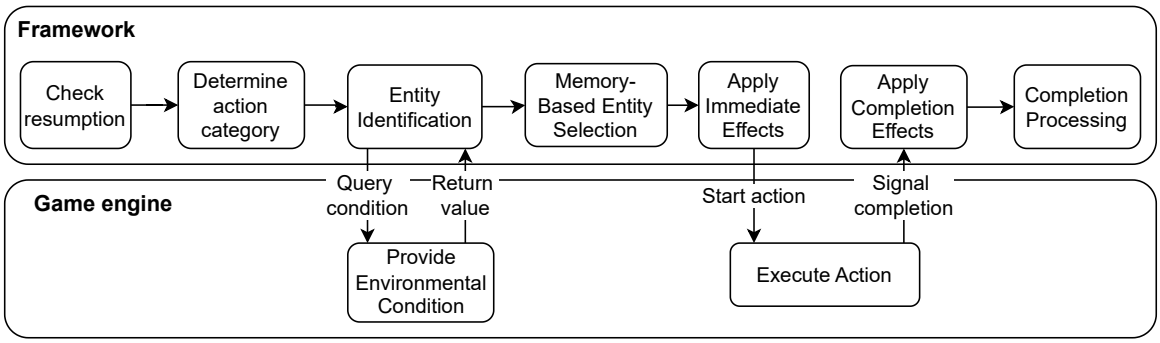

Figure 3.7: Action execution flow. Each row corresponds to a system: The framework (top) handles decision logic, and the game engine (bottom) handles action execution.

Before executing an action from scratch, the framework checks whether the action was previously interrupted. If a matching interruption memory exists and its preserved context is still valid, the framework resumes the action rather than restarting it. This resumption process is described in Section 3.3.5. The remainder of this section describes the case where no prior interruption exists.

The framework first checks whether the action requires interaction with another entity. Independently-performed actions (such as resting) need no entity selection, so the framework proceeds directly to effect application. For entity-requiring actions (such as sitting on a bench or talking to another NPC), the framework must first identify and select a suitable entity.

Entity Identification

Entity identification involves two filtering steps. First, the framework identifies all entities that declare support for the required action. Second, it evaluates each candidate against the action's preconditions using state operations (Section 3.1.2). Only entities that both support the action and satisfy all preconditions are selected.

Continuing the park example, suppose the visitor NPC reaches the "sit on bench" action node. The framework finds three park benches that support the action, then evaluates each against the preconditions: The bench must have available capacity and the weather must not be rainy. If "Park Bench B" is fully occupied, it fails the capacity check, leaving "Park Bench A" and "Park Bench C" as valid candidates.

When multiple entities remain after filtering, the framework applies the memory-driven selection algorithm (Section 3.1.5) using the NPC's action memories. If an action memory records that "Park Bench A" was recently used for "sit on bench" but no memory exists for "Park Bench C," the algorithm selects "Park Bench C." When only a single entity satisfies all preconditions, the framework selects it directly.

If no entities support the required action or none satisfy its preconditions, the action cannot proceed. The framework puts the sequence in the failed state and activates the recovery mechanisms described in Section 3.3.6.

Effect Application and Decision Recording

Once an entity has been selected (for entity-requiring actions) or immediately for independently-performed actions, the framework applies the action's immediate effects. These state changes occur before the action becomes visible in the game, ensuring that other NPCs see accurate state when evaluating the same entities. For the "sit on bench" action with "Park Bench C" selected, the bench's occupied seats count increases and the NPC's sitting state is set to true. The framework then creates an action memory recording that "Park Bench C" was used for this action.

After immediate effects are applied and the action memory is recorded, the framework signals the game engine to begin rendering the action. The game engine receives a reference to the NPC, the action to perform, and the target entity (if applicable). The sequence remains in the waiting for action state until either the game engine signals completion or the action's maximum timeout elapses (Section 3.1.3).

When the action completes, the framework applies the action's completion effects (e.g., increasing the visitor NPC's energy after resting). The sequence then transitions to the node executed state, and the framework evaluates outgoing transitions as described in Section 3.3.2.

3.3.4 Interruption Handling

Game events sometimes require NPCs to stop what they are currently doing and respond to something more urgent. A sudden rainstorm might require park visitors to seek shelter, a fire alarm might require building occupants to evacuate, or a player's actions might provoke

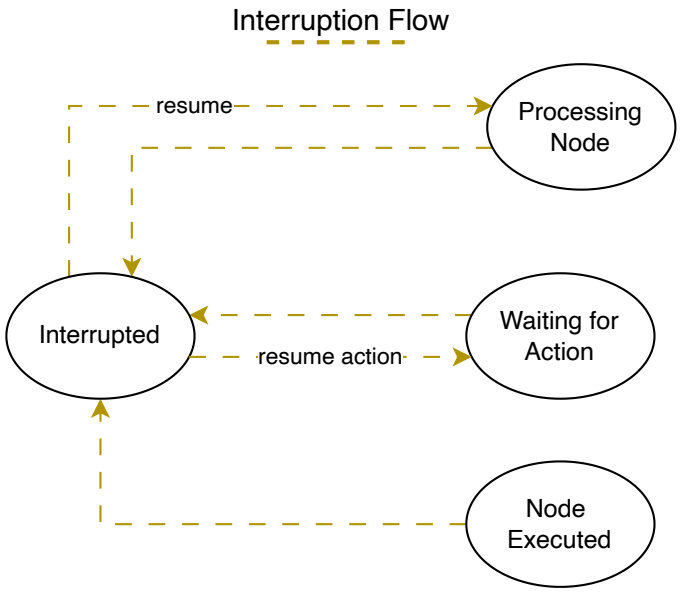

Figure 3.8: Interruption Flow.

nearby NPCs to react. This section describes how the framework handles these interruptions with Figure 3.8 showing the full interruption flow.

Handler Verification

When a game event triggers an interruption, the framework checks whether the affected NPC has a registered response for that event type. Each NPC defines **interruption handlers**, which are configured mappings from interruption types to response behavior sequences. If an NPC has no mapping for an event, the interruption is ignored. This means different NPCs can respond differently to the same event: A park maintenance worker might have a response for a "park closing" event that a visitor NPC does not.

Continuing the park example, suppose the visitor NPC is executing the "sit on bench" action on "Park Bench C" when a rainstorm begins. The game's weather system triggers a "rainstorm" interruption. The framework checks the visitor NPC's interruption handlers and finds a mapping from "rainstorm" to a "seek shelter" response behavior sequence.

Context Preservation

Before interrupting the current action, the framework checks the action's resumability property (Section 3.1.3). If the action is resumable, the framework creates an interruption memory (Section 3.1.5) storing the context needed to continue the action later. If the action is non-resumable, no interruption memory is created.

The framework then applies the action's interruption effects to maintain state consistency.

For the "sit on bench" action, the bench's occupied seat count decreases and the NPC's sitting state is set to false, freeing the bench for other NPCs while the visitor seeks shelter.

After context is preserved and interruption effects are applied, the framework transitions the current behavior sequence to the **interrupted** state and pushes the response behavior sequence onto the NPC's sequence stack (as shown in Figure 3.9). The response sequence becomes the active sequence and executes normally (Section 3.3.2). For the visitor NPC, the "seek shelter" sequence directs the NPC to a nearby covered area. Once the response sequence completes and is removed from the stack, the interrupted behavior sequence becomes active again and the framework attempts to resume from where the NPC left off.

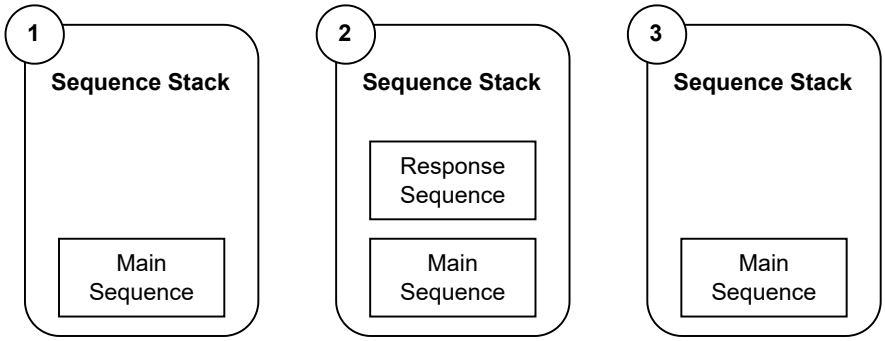

Figure 3.9: Example of interruption affecting a sequence stack.

3.3.5 Resumption

The framework must now determine whether the NPC can continue from where it was interrupted or whether conditions have changed too much to resume.

If the behavior sequence was in the waiting for action state when interrupted (meaning an action was in progress), the framework searches the NPC's interruption memories for a record matching the current action, sequence, and node. If a match is found, the framework verifies that the preserved context is still viable: The referenced entity must still be available (for entity-requiring actions) and the action's preconditions must still be satisfied under current game conditions.

Successful Resumption occurs when the preserved context is still valid. The framework removes the interruption memory, re-applies the action's immediate effects (restoring the state changes that were reversed by the interruption effects), and signals the game engine to continue the action. The sequence returns to the waiting for action state. For the visitor NPC, if the rainstorm has passed and "Park Bench C" is still available with its preconditions satisfied, the NPC returns to sit on the same bench.

Failed Resumption occurs when conditions have changed during the interruption. The framework discards the interruption memory and attempts to execute the action from scratch following the process in Section 3.3.3. If other valid entities exist (perhaps "Park Bench A" has become available), the action proceeds with a new entity. If no valid entities exist, the sequence enters the failed state and recovery mechanisms activate (Section 3.3.6).

If the behavior sequence was not executing an action when interrupted (e.g., it was in the processing node or node executed state), no action resumption is needed. The sequence continues from its current position.

3.3.6 Behavioral Resilience and Fallbacks

A behavior sequence enters the **failed** state when any of the following occurs: No outgoing transitions from the current node have their preconditions satisfied (Section 3.3.2), an entity-requiring action cannot proceed because no entities satisfy its preconditions (Section 3.3.3), or an interrupted action cannot be resumed and no alternative entities are available (Section 3.3.5). These situations, which we call **behavior failures** arise naturally as game conditions change and are not software errors. Figure 3.10 shows how these "failures" affect the flow of a sequence.

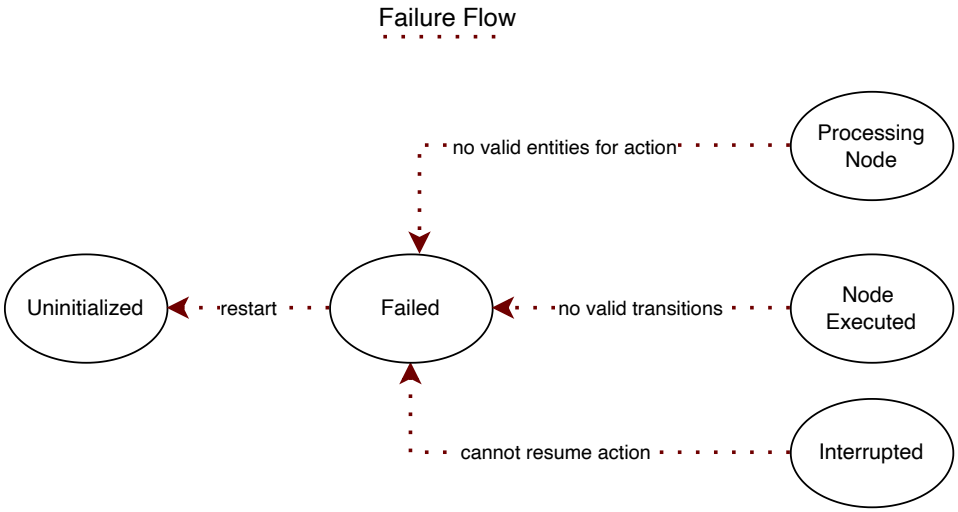

Figure 3.10: Failure Flow.

Fallback Sequence Activation

Each NPC can be configured with one or more fallback behavior sequences that serve as behavioral safety nets. When a behavior sequence enters the failed state, the framework clears all interruption memories associated with that sequence (preventing stale context from interfering with future executions), removes the failed sequence from the stack, and pushes a fallback behavior sequence in its place (as shown in Figure 3.11). When multiple fallback sequences are available, the framework selects one randomly to provide variety even during recovery. The fallback sequence executes normally from its entry node. Once the fallback completes, the stack is empty (since the failed sequence was also removed), and the NPC's main behavior sequence restarts automatically (Section 3.1.4). For example, if all benches are occupied and the visitor NPC's "park visitor" sequence fails, a fallback such

as “idle wander” (standing idle, walking to a nearby point) keeps the NPC active until the main sequence restarts under potentially different game conditions.

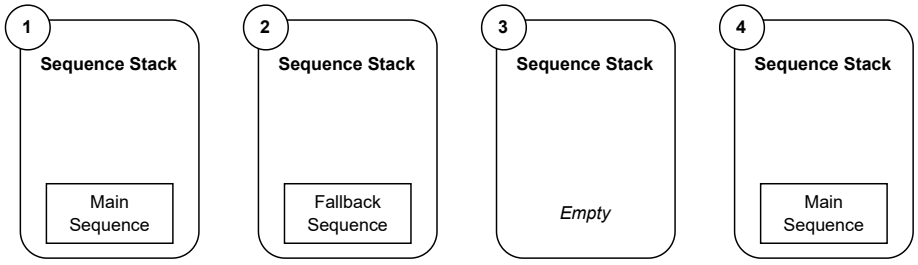

Figure 3.11: Example of a behavior failure affecting a sequence stack.

Fallback Sequence Design Requirements

For the recovery mechanism to function reliably, fallback behavior sequences must satisfy specific design constraints. Their actions and transitions should have no preconditions, and actions should be independently-performed (not requiring specific entities), ensuring execution can proceed regardless of game conditions. The behaviors should be appropriate defaults for the NPC (e.g., idle animations, simple movements, or basic activities) that maintain believability while the NPC transitions between sequences. Fallback sequences must be designed to complete successfully under all conditions, since a failure within a fallback would trigger the consecutive failure safeguard, ultimately halting the NPC’s behavioral processing entirely.

Consecutive Failure Safeguard

Despite careful fallback design, misconfigured behavioral content could create situations where fallback sequences fail or where main sequences immediately fail upon restart, creating a continuous failure-recovery cycle. To protect against this, the framework tracks consecutive failure activations for each NPC. If an NPC exceeds a configurable threshold of consecutive failures without completing at least one action successfully, the framework halts that NPC’s behavioral processing. This prevents a misconfigured NPC from consuming computational resources in an infinite loop and serves as a signal to developers during testing that the NPC’s behavioral content needs adjustment.

3.4 Framework Scope and Integration Requirements

This section explains what our framework provides and what it requires from game developers who adopt it.

Framework Scope

Our framework addresses one specific concern within the broader challenge of ambient NPC simulation: behavioral decision-making. It determines what an NPC should do next and which entities it interacts with, but does not determine how actions are rendered within a game. This separation, established in Section 3.1.3, ensures that the framework avoids dependencies on any particular game engine technology.

External System Requirements

Game developers integrating our framework must provide four categories of external functionality.

The first is **environmental context provision**. As described in Section 3.1.2, our framework periodically requests values for game environmental conditions (e.g., weather, time of day) and caches them for use in state operation evaluations. Games must provide a mechanism to supply these values when requested.

The second is **action presentation**. When the framework determines that an NPC should perform an action, game engine systems must coordinate the appropriate visual and auditory execution: Playing animations, executing pathfinding, triggering audio, or whatever an action requires in a particular game. Game engine systems must also signal the framework when an action is complete.

The third is **entity lifecycle management**. The framework maintains references to entities (tracking which benches, stalls, and NPCs exist), while a game engine manages their spatial presence within the game (spawning, positioning, and removing entities).

The fourth is **behavioral specifications**. Developers must define the behavioral specifications for their NPCs: The actions available in the game, the behavior sequences that organize those actions, the properties that entities maintain, the environmental conditions the framework should monitor, and the specific configurations for each entity (initial state, supported actions, and, for behavioral entities, their main behavior sequence, fallback sequences, interruption handlers, and memory capacities). These specifications represent the creative design work that gives each game its distinctive NPC behaviors.

Table 3.3 summarizes this division of responsibilities between the framework and external systems.

Data-Driven Architecture

As introduced at the beginning of this chapter, our framework operates on behavioral specifications rather than hard-coded behavioral logic. Actions, behavior sequences, entity configurations, and environmental condition definitions are all specified outside of the framework's code. This means that adding a new action, adjusting an NPC's behavior sequences, or changing the conditions that influence decisions requires only changes to these specifications, not to framework code. This data-driven approach also enables iterative development: Designers can adjust behavioral content and observe the results without recompilation. The specific format for defining behavioral content is described in Chapter 4. Once integration with the framework is complete (through the four external system requirements described

Responsibility	Framework	External Systems
Behavioral Decision-Making		
Manage entity state	✓	
Select next action	✓	
Evaluate preconditions	✓	
Apply memory selection	✓	
Control sequence execution	✓	
Handle interruptions	✓	
Handle behavioral "failures"	✓	
Content and State		
Define behavioral specifications		✓
Provide game environmental conditions		✓
Maintain entity lifecycle		✓
Signal interruptions		✓
Action Presentation		
Animate actions		✓
Produce audio-visual effects		✓
Signal action completion		✓

Table 3.3: Framework vs External Systems Responsibilities

above), the development effort shifts from implementing behavioral algorithms to designing behavioral content: Defining actions with appropriate preconditions and effects, composing behavior sequences that create interesting behavioral patterns, and configuring NPCs with suitable main sequences, fallbacks, and interruption handlers.

4. Framework Implementation

4.1 Implementation Design

This chapter presents an implementation of our framework that realizes the model described in Chapter 3 through concrete data structures, algorithms, and software architecture. Our implementation is written in C++ and packaged as a Dynamic Link Library (DLL), which is used in both Unity and Unreal Engine to validate the engine-agnostic design.

4.1.1 Design Goals

The following goals guided the implementation decisions described in this chapter:

- **Engine-agnostic architecture**: Our framework's behavior planning and decision-making components must operate independently of any specific game engine. Integration with a game engine should require only a thin coordination layer.
- **Data-driven configuration**: All behavior specifications, including action definitions, behavior sequence definitions, and entity state configurations, are defined in external configuration files and loaded at runtime. This enables developers to modify NPC behavior without modifying our framework's implementation.
- **Separation between behavior logic and animations**: Our framework selects which actions NPCs should perform, while game engine systems handle the visual and auditory presentation of those actions (as established in Section 3.1.3).

4.1.2 Technology Selection Rationale

To support engine-agnostic integration, our framework is distributed as a **Dynamic Link Library (DLL)**. This allows it to be compiled into a standalone binary that can be loaded at runtime by any compatible application. Two implementation languages were considered: C# (.NET) and native C++. A .NET implementation would have been directly interoperable with Unity, which uses C#. However, integrating .NET code with Unreal Engine's native C++ runtime required substantial engineering effort disproportionate to the scope of this project. A C++ DLL integrates natively with Unreal Engine and can be invoked from Unity using Platform Invocation Services (P/Invoke), a .NET mechanism that allows managed C# code to call functions in unmanaged native libraries. Documentation review of other engines, including Godot, suggested that most engines can load native C++ DLLs, whereas direct support for .NET assemblies is less widely available. For these reasons, C++ was selected as the framework's implementation language.

Although our framework is implemented in C++, the functionality needed to integrate with game engines is exposed through a **public C API** using an `extern "C"` interface

rather than via C++ classes. This avoids C++ name mangling and Application Binary Interface (ABI) incompatibilities, which can arise when C++ libraries are used across different compilers or programming languages. By relying on the stable and widely supported C calling convention, our framework maximizes portability and can be integrated with nearly all programming languages and game engines.

Our framework uses a **CMake-based build system** that generates platform-appropriate build files for both the DLL and a static library used for internal testing. Unit tests were implemented using the Google Test framework, covering all framework components described in Chapter 3 and the services described in Section 4.3.

Our framework's data-driven design requires **configuration files** that developers can create and edit. JSON was selected for its concise and well-known syntax, its practical balance between readability and parsing simplicity compared to alternatives such as XML or YAML, and the availability of the nlohmann/json library for straightforward C++ integration. Configuration files are small (typically a few kilobytes each) and are loaded once during framework initialization, so parsing performance is not a concern. The configuration file types and loading process are described in Section 4.2.4.

4.2 Data Structures and Algorithms

This section describes the data structures chosen for our implementation and the reasoning behind each choice, organized by the three component groups defined in Chapter 3 (Entity, Behavioral, Memory). Before describing each group, we describe two design conventions that affect data structures throughout the framework.

Integer representation: All identifiers (such as action, behavior sequence, entity, and state key identifiers) and entity state values in our framework use **32-bit signed integers**. Signed integers were chosen over unsigned integers because they simplify debugging (negative values are immediately recognizable as erroneous, whereas underflow in unsigned integers can produce large positive values) and because they allow reserving negative values for special conditions. For example, the current action identifier in a behavioral entity uses -1 to indicate that no action is active, allowing a single variable to represent both valid identifiers and the absence of a value. Some values, such as memory record timestamps and action durations, use **64-bit signed integers** to represent time in milliseconds, accommodating games that run for extended periods. Integers were chosen over alternatives such as floating-point numbers or strings because the values our framework represents (identifiers, capacity counts, energy levels, enumerated conditions) are discrete and bounded, and integer comparisons and arithmetic are computationally inexpensive.

Opaque handles: To reference entities managed by game engines without depending on engine-specific object representations, our framework uses **opaque handles** stored as `void*` pointers. When a game engine adds an entity to our framework, the game engine provides a handle that identifies that entity within its own systems. Our framework stores this handle and passes it back to the game engine unchanged in all callbacks, ensuring that our framework

never needs to examine or interpret the handle's content. Each registered entity also receives an internal integer identifier, defined in its configuration file, that our framework uses for all internal operations. The mapping between these identifiers and the opaque engine handles is maintained through lookup tables described in Section 4.3.1.

4.2.1 Entity Group

Figure 4.1 shows the UML class diagram of the entity group in our framework.

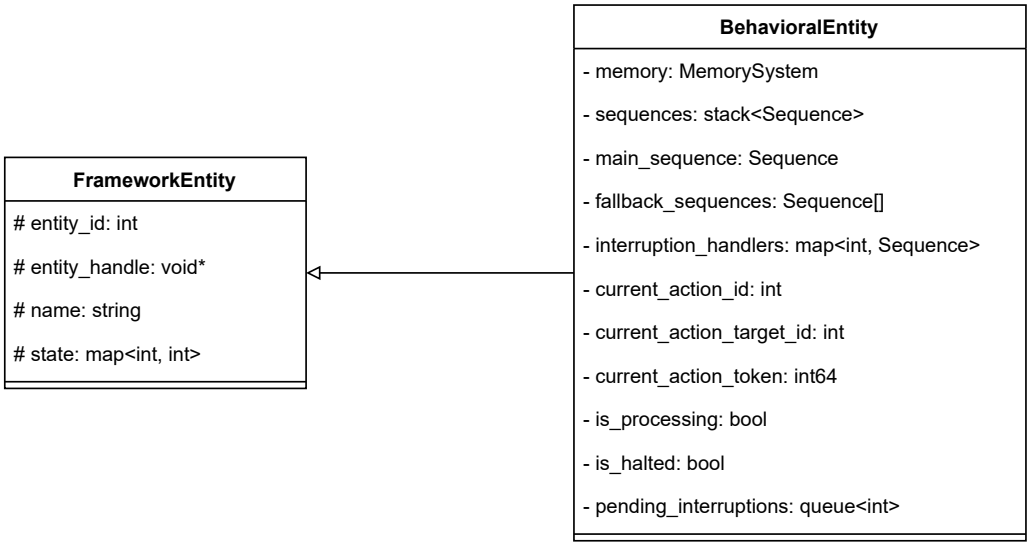

Figure 4.1: Entity Group Class Diagram.

Framework entities are represented by a class that stores four properties: the opaque engine handle (described above), a non-negative integer identifier, an optional name used for logging and debugging, and a hash map of state properties mapping integer keys to integer values. The hash map was chosen because precondition evaluation requires retrieving specific state properties by its integer key, and this retrieval occurs at every decision point in an NPC's execution. Hash maps provide average O(1) access, ensuring that individual state lookups remain constant-time regardless of how many properties an entity has. The integer keys used to identify state properties at runtime are called **state keys**. The mapping between human-readable names in configuration files and these integer keys is maintained by the schema manager service (described in Section 4.3.1), enabling developers to work with meaningful names such as "CAPACITY" while the framework operates with integers. Information about which actions an entity supports is not stored on the entity itself but is maintained by the framework registry (Section 4.3.1), which keeps bidirectional lookup tables mapping actions to supporting entities and entities to supported actions.

Behavioral entities extend the framework entity representation with the properties needed for NPC decision-making as described in Section 3.1.1. These properties fall into three categories: Memory and behavior specifications, action execution context, and flow control.

For memory and behavior specifications, each NPC owns a memory system (described in Section 4.2.3) and manages its behavior sequences through a sequence stack. Each NPC also stores references to its main behavior sequence, its fallback sequences (Section 3.3.6), and a hash table that maps interruption identifiers to response behavior sequence identifiers, providing average O(1) handler lookup when interruptions are triggered.

For action execution context, each NPC tracks the identifier of its current action, the identifier of the entity being used (or a value of -1 for independently-performed actions), and an action token. The action token is a **64-bit integer** that increments each time an NPC begins a new action. When the framework signals a game engine to start an action, it includes the current token value. When the engine later signals completion, it must return this token. If the token returned by the engine does not match the token stored in the framework (for example, because the action timed out and a new action began in the interim), the completion signal is rejected.

For flow control, each NPC maintains a processing flag that ensures its execution loop cannot be triggered again while it is already running, a halted flag that stops further updates after the NPC exceeds a maximum number of consecutive fallback attempts (set to three in our implementation), and a queue of pending interruptions. Interruptions can accumulate in two ways: multiple game events may occur before the NPC's next update, or a new interruption may arrive while the NPC is still processing a previous one. When the framework updates an NPC, it processes any pending interruptions from the queue before evaluating any transitions or actions in the NPC's current behavior sequence, ensuring that interruption handling always operates on a stable sequence state. For example, a "rain" event may interrupt an NPC mid-sequence and push a shelter-seeking behavior sequence onto the sequence stack, and while that sequence is executing, a "fire" event may cause a second interruption. Once the fire interruption sequence completes and is removed from the stack, the NPC resumes the rain interruption sequence, which remains on the stack.

4.2.2 Behavior Group

State operations (defined in Section 3.1.2) are represented by four integer fields:

- A **target identifier** that specifies what the operation applies to: The NPC itself, another entity, an environmental condition, or the distance between two entities.
- A **state key** identifying which property to access on the target.
- An **operation type** determining whether the operation is a comparison or a modification.
- A **value** specifying the threshold for comparisons or the amount for modifications.

Figure 4.2 shows the UML class diagram of state operations. Distance between two entities is computed using Manhattan distance from positional coordinates provided by a game engine. Although Euclidean distance is more accurate, our framework only performs threshold-based comparisons (for example, checking whether the distance is below a certain value). For such

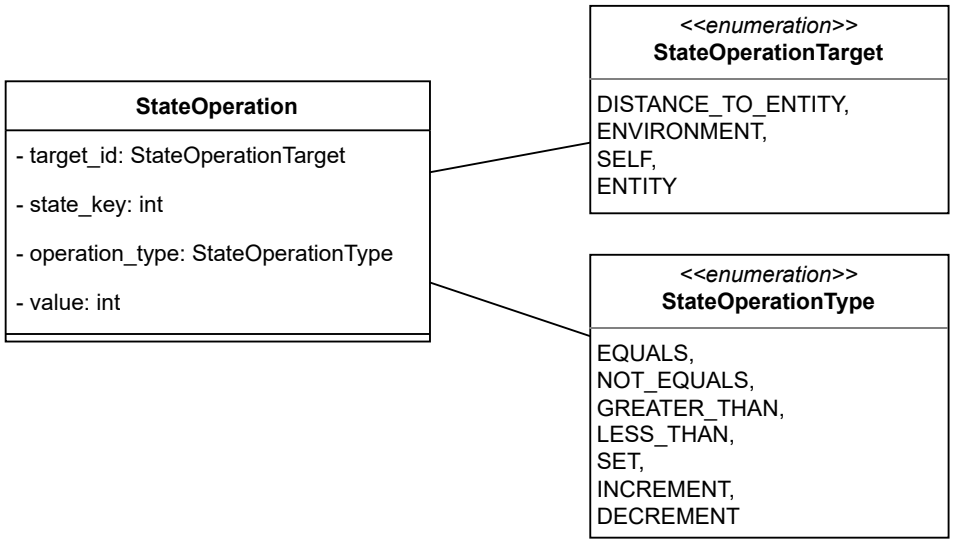

Figure 4.2: State Operation Class Diagram.

comparisons, Manhattan distance provides a sufficient approximation while avoiding floating-point arithmetic and remaining consistent with the integer-based representation described earlier. Entity positions are not part of the entity state hash map described in Section 4.2.2. Instead, they are periodically queried from the game engine and cached by the entity position manager (Section 4.3.1), with a configurable refresh frequency per entity. Distance-based preconditions evaluate against these cached values rather than reading from stored entity states.

Comparison operations are used in precondition evaluation, while modification operations are used in effect application. The operation type encoding reserves a range of values for game-specific operations, providing an extension point for developers who need custom operations beyond the built-in set. The mechanism for registering and processing such custom operations is left as future work.

Behavior sequences (defined in Section 3.1.4) are represented with three structures as shown in Figure 4.3: A hash map of nodes indexed by node identifier, an adjacency list of transitions indexed by source node identifier, and a set of execution state fields. The adjacency list groups transitions by their source node so that when the framework needs to evaluate which transitions are available from the current node, it performs a single hash map lookup to retrieve all outgoing transitions. This is relevant because transition evaluation occurs at every decision point in each NPC's execution cycle.

Each behavior sequence tracks its lifecycle state (Section 3.3.1), a reference to the current node being processed, and an entry point node identifier used when resetting the sequence. A behavior sequence definition is loaded once from its JSON-defined configuration file. When an NPC is assigned a behavior sequence, the framework creates a copy of the behavior sequence for the NPC, allowing multiple NPCs to follow the same behavior sequence simultaneously while each progresses through it independently.

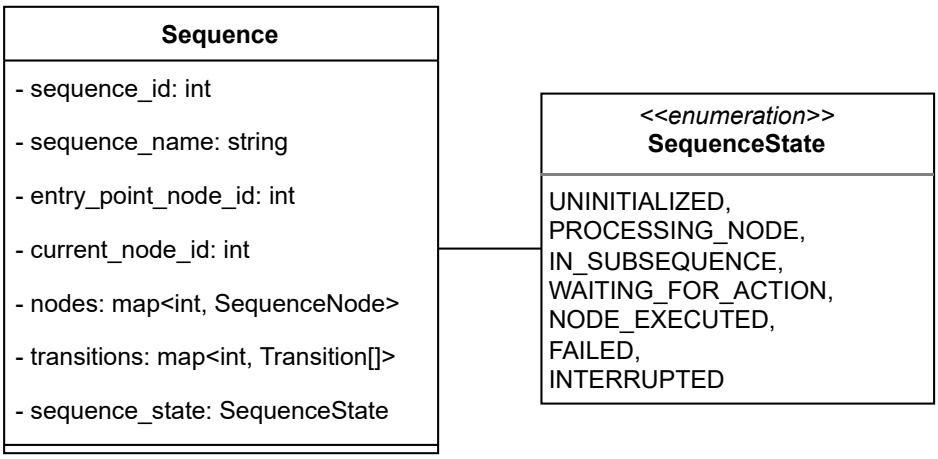

Figure 4.3: Behavior Sequence Class Diagram.

Sequence nodes correspond to the three node types defined in Section 3.1.4 as shown in Figure 4.4: Action nodes store the identifier of the action to execute, nested sequence nodes store the identifier of the sequence to push into the NPC's sequence stack, and end nodes contain no additional data. All three sequence node types share a completion flag that records whether the action or behavior sequence represented by a node has been executed during the current pass through the sequence. When a sequence revisits a node (either after restarting due to a behavioral failure as described in Section 3.3.6, or when a transition loops back to an earlier node), the completion flag is reset so that results from a previous execution of the sequence does not influence the current one.

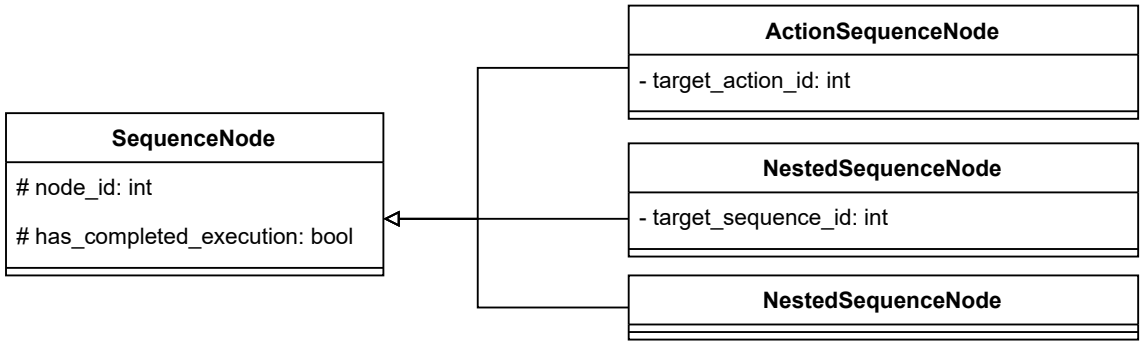

Figure 4.4: Sequence Nodes Class Diagram.

Transitions (shown in Figure 4.5) store an identifier, a destination node identifier, and a collection of preconditions implemented as state operations. The preconditions are stored in a hash map indexed by target identifier, grouping them by whether they apply to the NPC, a specific entity, or an environmental condition. This allows the state operation evaluator to retrieve only those preconditions relevant to a specific target during evaluation, avoiding iteration over all preconditions associated to the transition.

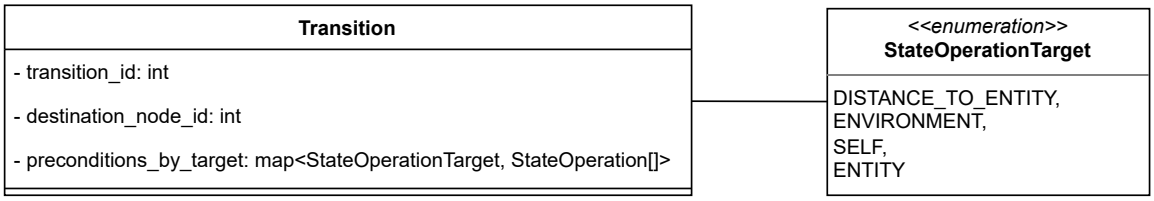

Figure 4.5: Transition Class Diagram.

Actions store the behavioral properties defined in Section 3.1.3 as shown in Figure 4.6. Preconditions are organized using the same hash map structure as transitions, grouping preconditions by target identifier and enabling the same evaluation process. Immediate effects, completion effects, and interruption effects are stored as vectors of state operations.

Each action stores its expected duration and maximum timeout as **64-bit integers** representing time in milliseconds, and a resumability flag indicating whether the action can be resumed after interruption (Section 3.3.4). Whether an action requires a target entity is determined at runtime by checking if any of its preconditions reference a specific entity or the distance to a specific entity. If so, the action follows the entity identification process described in Section 3.3.3.

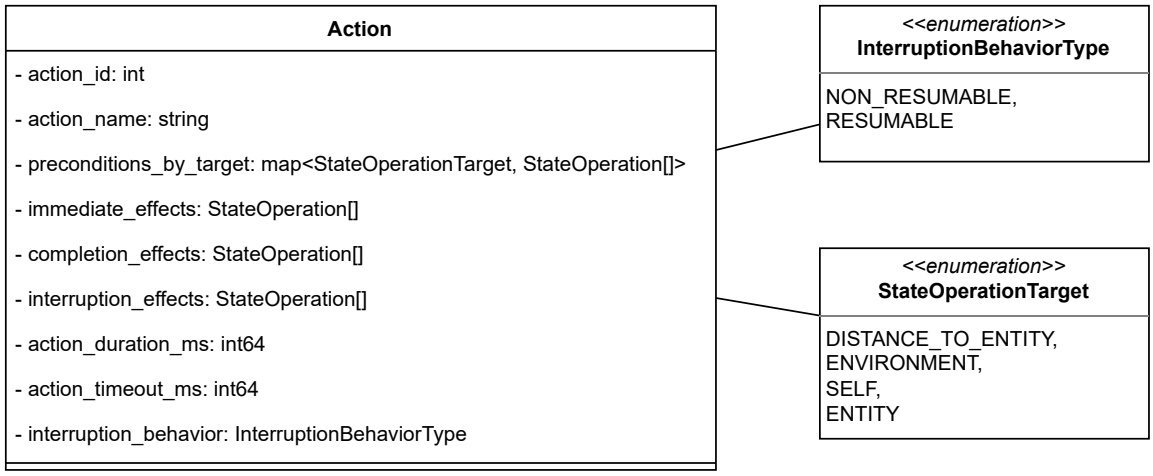

Figure 4.6: Action Class Diagram.

4.2.3 Memory Group

Each NPC's memory system stores three types of memory records (transition, action, and interruption) in separate containers, each with a configurable maximum capacity that limits how many past records an NPC retains. The containers use a first-in, first-out pattern for capacity enforcement, always removing the oldest record when capacity is reached. However, when a new record matches a decision already in the container, the existing record must be removed before the new one is inserted at the back of the container. Updating the existing

record in place would not be sufficient because capacity enforcement assumes that earlier positions hold older records; an in-place update would leave a recently updated record in an early position, causing it to be removed before older records.

The chosen container structure (a deque) provides unrestricted element access (which the deduplication step requires) and stores elements in contiguous memory blocks, offering good cache performance for the linear scans performed by the memory-driven selection algorithm. This design allows the oldest record to be removed from the front in average O(1) when capacity is reached, at the cost of average O(n) insertion due to the linear search required for deduplication. Given that memory capacities remain small in practice (typically fewer than 20 entries per type per NPC in the experiments presented in Chapter 6), the average O(n) insertion is reasonable.

As shown in Figure 4.7, all memory record types store a creation timestamp as a 64-bit integer. Each type stores additional fields that identify the specific decision the record represents, as defined in Section 3.1.5. During interruption memory lookup, when the framework searches for an existing interruption memory, it matches the action identifier, behavior sequence identifier, and node identifier. The entity identifier is stored in the record as context for the resumption process but is not used for matching, because the same action at the same node in the same behavior sequence represents the same interruption context regardless of which entity was involved.

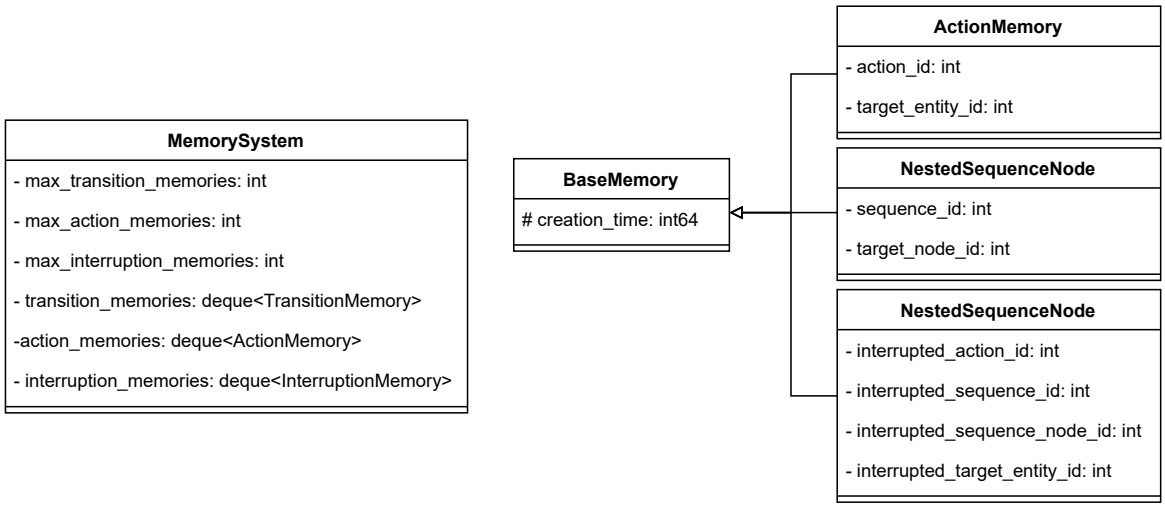

Figure 4.7: Memory Group Class Diagram.

Algorithm 1 presents the memory-driven selection algorithm used at two types of decision points: selecting which transition to follow within a behavior sequence and selecting which entity to interact with when executing an entity-requiring action. The algorithm combines exploration and least-recently-used selection: it first identifies candidate options that have no corresponding memory record and selects randomly among them (lines 1–13), encouraging exploration of new alternatives. If all candidates have been previously used, the algorithm identifies those with the oldest creation timestamps (lines 14–23) and randomly selects among them to prevent deterministic patterns (line 24). When used for transition selection, the

Algorithm 1 Memory-Driven Action Selection

Require: *options*: valid transition destinations or candidate entities, *memories*: memory records storing previously selected options
Ensure: ID of the selected option

```
unusedOptions ← empty list
usedMemories ← empty list
for all option in options do
    record ← memory record in memories matching option
    if record does not exist then
        append option to unusedOptions
    else
        append record to usedMemories
    end if
end for
if unusedOptions is not empty then
    return random choice from unusedOptions
end if
oldestTime ← ∞
candidates ← empty list
for all record in usedMemories do
    if record.time < oldestTime then
        oldestTime ← record.time
        candidates ← list containing only record.optionID
    else if record.time = oldestTime then
        append record.optionID to candidates
    end if
end for
return random choice from candidates
```

algorithm matches transition memories on behavior sequence identifier and destination node identifier, and `record.optionID` refers to a destination node identifier. When used for entity selection, it matches action memories on action identifier and entity identifier, and `record.optionID` refers to an entity identifier.

4.2.4 Configuration System

Our framework uses five types of configuration files. Table 4.1 summarizes each file type, its purpose, and when it is loaded. Examples of each configuration file format and a full description of their fields are provided in Appendix A.

Configuration files are loaded during framework initialization in a fixed order: state schema, environmental conditions, actions, and behavior sequences. Entity configuration files are loaded later at runtime when each entity is registered (described in Section 4.4.2). This ordering is necessary because each file type references identifiers defined by previously loaded files: actions reference state keys from the schema, behavior sequences reference action

File Type	Purpose	Loaded
State schema	Defines mappings between human-readable state names and integer keys used at runtime	Initialization
Environmental conditions	Defines game-world conditions available for precondition evaluation (e.g. weather, time of day) and their update frequencies	Initialization
Actions	Defines all actions with their preconditions, effects, and execution parameters	Initialization
Behavior Sequences	Defines all behavior sequences with their nodes, transitions, and entry points	Initialization
Entity configuration (per entity)	Defines an entity's properties, initial state, supported actions, and for behavioral entities, their sequence assignments and memory capacity limits per type.	Runtime (per entity registration)

Table 4.1: Configuration files and their responsibilities in our framework

identifiers, and entity configurations reference both behavior sequence and action identifiers. The framework validates these references during loading and reports errors through the logging service if any referenced identifier has not been previously registered.

4.3 DLL Architecture

The data structures described in Section 4.2 must be initialized, populated, and updated as NPCs execute their behavior sequences. This section describes how our DLL organizes these responsibilities. Figure 4.8 shows the overall architecture: our framework is packaged as a C++ DLL that contains all NPC decision-making systems, with game engine systems communicating through the public C API (described in Section 4.4.2) and all behavioral specifications provided through JSON-defined configuration files.

Figure 4.9 shows the internal organization of the DLL. We organize functionality into two categories: services, which manage framework infrastructure and state, and component groups, which implement the NPC decision-making processes described in Chapter 3. Services are grouped into layers based on their architectural role (described in the next section). A fourth component group, the integration group, handles the coordination of the other groups at runtime and communication between the framework and game engines through the public C API. This group is described in Section 4.4.

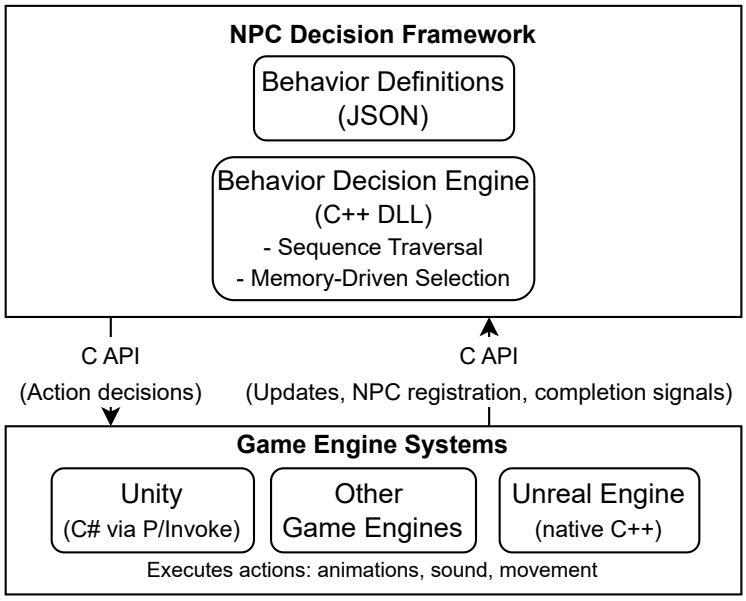

Figure 4.8: Engine-agnostic framework architecture.

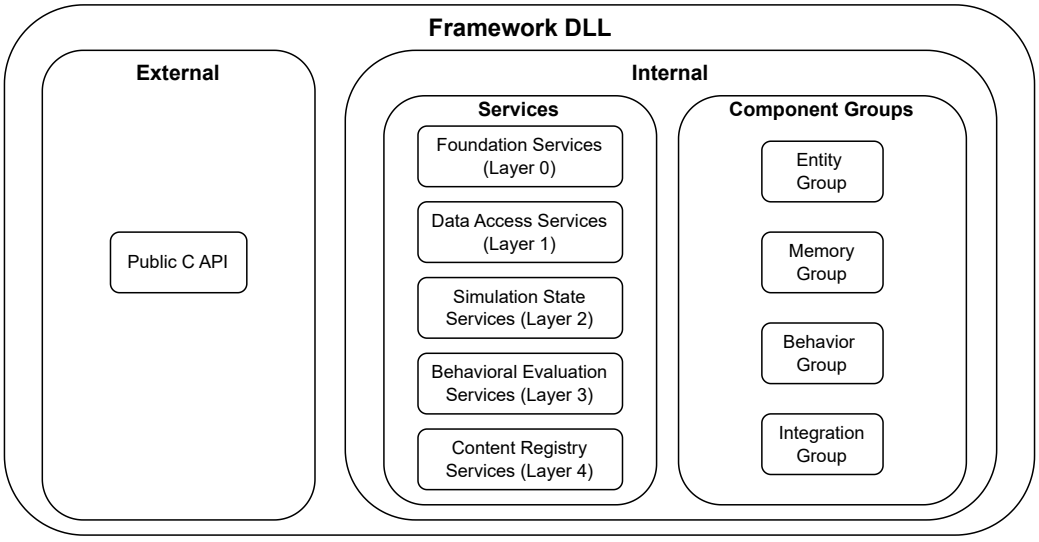

Figure 4.9: Organization of framework components in the DLL.

4.3.1 Service Layers

Service layers in our DLL follow a strict dependency hierarchy: a layer may depend on layers below it, but not on layers above it. Figure 4.10 shows the dependencies between services across all five layers. This unidirectional dependency flow provides three properties: (1) Services initialize in layer order (0 through 4), guaranteeing that all dependencies are satisfied before a service is constructed; (2) any layer can be tested with mock implementations of lower layers; and (3) circular dependencies are structurally impossible.

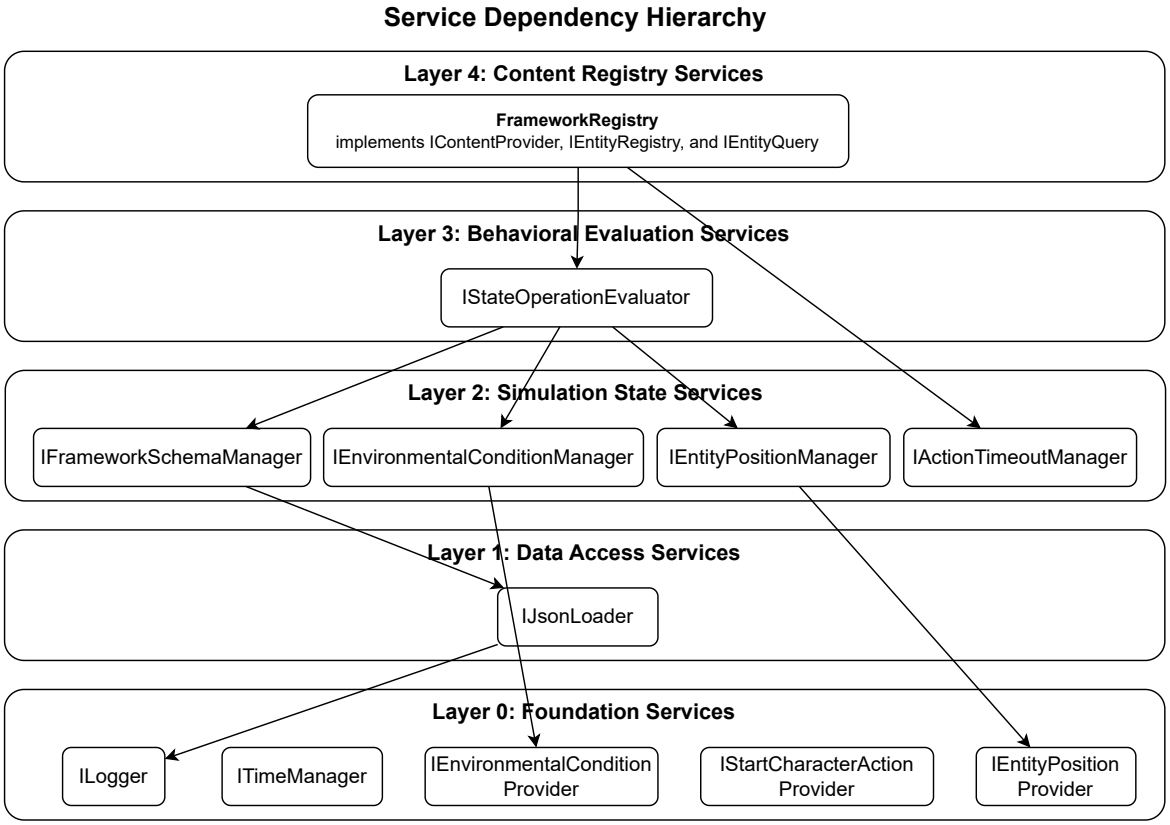

Figure 4.10: Service dependency hierarchy. An arrow from service A to service B indicates that A depends on B.

Foundation Services (Layer 0): These services provide core infrastructure with no dependencies on other framework services. This layer includes logging, time management, and three callback providers that wrap engine-provided function pointers into C++ interfaces. The callback providers handle environmental condition queries, action execution requests, and entity position queries. Wrapping these function pointers behind interfaces isolates the rest of the framework from engine-specific calling conventions and allows unit tests to substitute mock implementations without requiring full integration with a game engine.

Data Access Services (Layer 1): These services handle reading configuration files and converting their contents into data structures the framework can use. The JSON loader service translates JSON file contents into internal data transfer objects (DTOs), intermediate structures that contain only the values read from configuration files without any decision-making logic. By introducing this intermediate representation, the framework avoids coupling its internal components directly to the JSON format, allowing alternative configuration formats to be supported in the future without modifying the services that consume the parsed data.

Simulation State Services (Layer 2): These services manage values that may change during gameplay and are updated during each framework update cycle (described in Section 4.4.1). The **schema manager** maintains mappings between human-readable names and internal integer keys for entity states, operation types, and interruption handlers, loaded once during initialization from the state schema configuration file. The **environmental condition manager** caches environmental condition values obtained from a game engine, using the configurable refresh intervals described in Section 3.1.2 to determine when a cached value is stale and must be re-queried. The **entity position manager** provides access to entity positions and computes Manhattan distances between entities, using a similar caching strategy with configurable update frequencies. The **action timeout manager** tracks the start time and configured maximum duration for each active action, triggering automatic completion during the update cycle for any action that has exceeded its timeout (as defined in Section 3.1.3).

Behavioral Evaluation Services (Layer 3): This layer contains a single service: the **state operation evaluator** described in Section 4.2.2. It serves as the integration point where schema information, game environmental conditions, entity position data, and entity state converge to enable the evaluation of state operations.

Content Registry Services (Layer 4): These services provide the catalog of all framework content and manage entity lifecycles. The **framework registry** implements three interfaces that expose different views of the same underlying data: (1) access to actions and behavior sequences loaded from configuration files, (2) entity lifecycle management (registration, unregistration, and lookup), and (3) queries for which entities support specific actions. The rationale for combining these responsibilities in a single component is described in Section 4.3.2.

During **entity registration**, the framework registry loads the entity's configuration file, creates the appropriate entity representation (framework entity or behavioral entity), assigns an internal integer identifier, records the mapping to the opaque engine handle, and updates the bidirectional action-entity lookup tables. For behavioral entities, the registry also instantiates private copies of their assigned behavior sequences and initializes the memory system with the configured capacity limits. **Entity unregistration** reverses this process, removing the entity from all collections, lookup tables, and position tracking. The public C API functions that trigger these operations are described in Section 4.4.2.

4.3.2 Service Layer Coordination

Service bundles simplify service access and clarify dependencies across layers. Each bundle is a struct that groups references to all services within a given layer. When a framework component is constructed, it receives the service bundle for its layer, giving it access to all services at that layer and below. This reduces constructor parameter lists and makes dependency chains explicit by reflecting the layer structure directly in the constructor signature.

The framework registry centralizes entity registrations, action mappings, and position tracking in a single component. While this combines multiple responsibilities, the design enforces **shared invariants**: When an entity is unregistered, the registry ensures that all associated data (entity collections, action-entity lookup tables, handle mappings, and position tracking) is removed consistently. Splitting these responsibilities across separate components would require additional synchronization logic to maintain this consistency. At the target NPC population scale of our experiments (hundreds of NPCs, see Chapter 6), the centralized design provides predictable performance. Larger implementations could benefit from decomposing the registry and introducing explicit synchronization strategies.

Service initialization proceeds in two phases. First, services in Layers 0 through 3 are created and grouped into bundles, establishing ownership and dependency relationships. Second, Layer 4 services are initialized, as they require access to the complete bundle hierarchy. This ensures that services always receive references to the services they depend on while preserving a clear initialization order.

Our framework's **threading model** executes entirely on the game engine's main thread, respecting Unity's single-threaded model for gameplay logic and Unreal's game thread restrictions. Performance evaluations in Chapter 6 show this approach is sufficient for hundreds of NPCs. Implementations targeting larger NPC populations could explore parallelization, such as partitioning entity updates across worker threads.

4.4 Framework Orchestration and Integration

The three component groups described in Section 4.2 handle entity state, behavioral logic, and memory. However, Chapter 3 deliberately left open how these groups are coordinated at runtime and how external systems communicate with the framework, since these are implementation concerns. Our implementation introduces a fourth component group, the **Integration Group**, to address these concerns. This group encompasses two responsibilities: internal orchestration (coordinating initialization, the update cycle, and event processing across all services and component groups) and external communication (providing the public C API through which game engines interact with the framework). The following sections describe each responsibility.

4.4.1 Framework Orchestration

A **central orchestration component** serves as the single entry point for all framework operations. It holds references to all services described in Section 4.3.1, manages framework

initialization, runs the update cycle, and processes events such as action completion signals and interruptions.

The **framework initialization** follows the service layer order described in Section 4.3.1. Foundation services are initialized first, followed by simulation state services, and finally the framework registry. Each stage loads the appropriate configuration files, validates references, and returns failure if any step does not succeed. If any stage fails, the framework aborts and logs failure details through the logging service.

The **update cycle** is shown in Algorithm 2. It receives the current time in milliseconds and a batch size. The batch size allows game engines to distribute NPC updates across multiple frames: a smaller batch size reduces per-frame processing time but increases the interval between consecutive updates for each NPC. A batch size of -1 processes all registered NPCs in a single cycle.

Algorithm 2 Framework Update Cycle

Require: *currentTime*, *batchSize*
Ensure: At most *batchSize* NPCs updated; next NPC index recorded
1: *frameworkTime* ← *currentTime*
2: **if** pending registration/unregistration commands exist **then**
3: process up to *batchSize* commands **return**
4: **end if**
5: mark timed-out actions as complete
6: *startIndex* ← index of next NPC to update
7: *count* ← min(*batchSize*, *totalEntities* − *startIndex*)
8: run one decision cycle for NPCs at positions [*startIndex*, *startIndex* + *count*)
9: *startIndex* ← (*startIndex* + *count*) mod *totalEntities*

The update method first synchronizes the framework's internal time. If any pending entity registration or unregistration commands exist, the method processes the number of NPCs specified by the batch size and returns immediately, deferring NPC updates to the next cycle. This ensures that the set of registered entities does not change while NPCs are being updated. When no pending commands exist, any actions that have exceeded their timeout are marked as complete. NPC updates then proceed in round-robin order, ensuring all NPCs are processed with roughly equal frequency. NPCs currently performing an action, halted by fallback mechanisms, or already being processed are skipped. A guard prevents overlapping update calls, which could corrupt entity state if a game engine triggered a second update before the first completed.

Entity registration and unregistration requests are queued rather than processed immediately for the consistency reason described above. The command queue also uses the batch size parameter, allowing developers to control how many entity operations are processed per cycle. During framework shutdown, a batch size of -1 is used to process all remaining commands.

4.4.2 External C API

Our framework exposes all functionality through the public C API described in Section 4.1.2. The API functions are divided into three categories: framework lifecycle, runtime operations, and entity management.

Framework lifecycle: The creation function accepts three engine callback function pointers (described in Section 4.4.3) and returns an opaque framework handle. The initialization function takes paths to the four configuration files (schema, sequences, actions, and environmental conditions) and a logging configuration (which defines the log file path and the log level), returning a boolean indicating whether all initialization stages succeeded. The shutdown function processes any remaining pending entity commands and releases all framework resources.

Runtime operations: The update function drives the update cycle described in Section 4.4.1, accepting the batch size and current time as parameters. The interruption processing function accepts an interruption identifier and an array of affected entity handles. For each handle, it looks up the corresponding NPC through the framework registry and triggers the interruption mechanism described in Section 3.3.4. Entities that are not behavioral entities or that lack a handler for the given interruption identifier are skipped.

Entity management: The registration function queues an entity for registration with its configuration file path and initial position (specified as three integer coordinate values). The unregistration function queues an entity for removal. Both operations take effect during the next update cycle as described in Section 4.4.1. The action completion function signals that the game engine has finished executing an action for a given entity, requiring the entity handle, action identifier, and action token. The framework validates that the action identifier and token match the NPC's currently active action before processing the completion, using the token mechanism described in Section 4.2.1.

4.4.3 Engine Integration Interfaces

The three callback function pointers provided at framework creation define the contract between our framework and a game engine. These callbacks are the only points where the framework depends on engine-provided functionality at runtime. Chapter 5 demonstrates concrete implementations of these callbacks in both Unity and Unreal Engine.

Start character action is invoked when the framework initiates action execution for an NPC. The framework passes the NPC's entity handle, the action identifier, the action token (Section 4.2.1), the expected action duration in milliseconds, and the target entity handle (or null for actions that do not require a target entity). The game engine is expected to activate the appropriate systems to execute the action (animations, navigation, audio) and must store the action token to return it when signaling action completion.

Query environmental condition is invoked when a cached environmental condition value becomes stale based on its configured update frequency (Section 4.3.1). The framework

passes the integer condition key, and the game engine returns the current value as a 32-bit integer.

Query entity position is invoked when the framework needs an updated position for an entity, typically during distance-based precondition evaluation. The framework passes the entity handle and a pointer to an output array of three 32-bit integers representing coordinates. The game engine writes the current position into this array and returns a boolean indicating success. A failed query causes the framework to use the previously cached position.

5. Integration with Game Engines

This chapter describes how the components from our framework described in Chapter 3 and Chapter 4 are integrated into two of the most popular commercial game engines: Unity and Unreal Engine. Each integration implements the coordination interface described in Section 4.4 (the three engine-provided callbacks and the public C API functions) while adapting to the specific conventions and systems of the host engine.

Each integration consists of two layers. The first is a reusable wrapper layer that implements our framework's coordination interface, manages entity registration and handle tracking, routes callbacks between the framework DLL and engine-specific objects, and provides abstract base classes for entities and a manager for the framework. This layer is designed so that developers adopting our framework can use it as a starting point for their own projects without having to develop this layer themselves. The second layer is a demo-specific layer that extends the wrapper layer's abstract base classes with concrete implementations for the demonstration scenarios used in our evaluation (described in Chapter 6). These implementations include the logic for supplying environmental condition values to the framework, the action execution logic, and the setup required in the visual development environment (called the **editor**) of each engine. The following sections introduce each game engine, describe the wrapper layer architecture and the demo-specific layer for each, explain the editor configuration required, and conclude with a comparison of the two integrations.

5.1 Unity Integration

5.1.1 Engine Overview

Unity is a widely used game engine that supports development across multiple platforms including Windows, macOS, game consoles, and mobile devices. Developers write gameplay logic in C#, which is referred to as the engine's scripting language. Unity runs C# code within the .NET runtime, which manages memory automatically through a garbage collector that periodically identifies and reclaims objects that are no longer in use.

Unity organizes games using a component-based architecture. Each object in a game is represented by a `GameObject` class, which serves as a container for components that provide functionality such as rendering, physics, navigation, and custom behavior. Components that need to respond to engine lifecycle events inherit from a base class, which provides event methods that the engine calls automatically: `Awake()` is called once when the game object initializes, `Start()` is called before the first frame, `Update()` is called every frame, and `OnDestroy()` is called when the game object is removed from the game.

Unity's gameplay execution is single-threaded, all component event methods execute sequentially on the main thread of a game within each frame. The engine does not guarantee the order in which different game objects have their events processed within a given frame, which requires careful design when coordinating behavior across multiple objects.

For NPC movement, Unity provides a built-in pathfinding system based on a navigation mesh (a data structure representing the walkable surfaces in a game). A navigation agent component attached to a character handles path computation and movement. For animation, Unity uses a state machine system where gameplay code drives transitions between animation clips.

Unity uses meters as its unit of measurement for positions and distances. This is relevant for the integration with our framework because distance values in the JSON configuration files are interpreted in the same coordinate space as the host engine.

5.1.2 Wrapper Layer Architecture

The Unity wrapper layer bridges our framework's public C API with Unity's C# environment. Since our framework is compiled as a native C++ DLL and Unity gameplay code is written in C#, the wrapper uses a language interoperability mechanism called Platform Invocation Services, or P/Invoke that allows C# code to call functions in native C and C++ libraries. The wrapper layer declares each function from the C API as a P/Invoke import, and at runtime the .NET runtime handles the type conversion between C# and C types when calls cross this boundary. This type conversion introduces a small overhead per call, which is measured as part of the engine-boundary cost in Chapter 6.

Handle Management

Our framework's C API uses opaque pointers to identify entities across callback invocations (as described in Section 4.4.2). In Unity, the garbage collector can relocate object in memory, which would invalidate such pointers. The wrapper layer addresses this by obtaining stable references for each registered entity that remain valid regardless of garbage collection activity. These stable references are stored in a bidirectional mapping that allows the wrapper to translate between framework handles and Unity objects in both directions: when sending entity handles to our framework during registration, and when resolving handles received from our framework during callbacks.

Callback Routing

The three engine-provided callbacks (defined in Section 4.4.3) are implemented as methods on the framework manager component. When our framework invokes a callback, the manager uses a hash table to map the entity handle to the corresponding Unity object and routes the call accordingly. For action requests, the manager maps the handle to a behavioral entity and forwards the action parameters. For position queries, it reads the entity's current position and returns the coordinates to our framework.

Abstract Base Classes

The wrapper layer provides three abstract base classes that developers must extend for their games. The framework manager base class handles framework creation, initialization with configuration file paths, per-frame updates, shutdown, and callback routing. Subclasses

must implement a method that creates the environmental query provider for their game. The ambient entity base class manages automatic registration with the framework during initialization and unregistration when an object is removed from the game, and stores the path to the entity's JSON configuration file. The behavioral entity base class extends the ambient entity base class with action execution support: It receives action requests from the manager, stores the current action identifier and action token, and provides a method for signalling action completion back to the framework. Subclasses implement the actual action execution logic for a game.

Configuration Asset

The wrapper layer defines a configuration data asset that holds all framework settings: paths to the four JSON configuration files (schema, sequences, actions, and environmental conditions), the entity configuration folder path, the log file path and log level, and the batch size parameter for our framework's update cycle. Developers can create an instance of this asset in the Unity editor and assign it to the framework manager object in their game.

5.1.3 Marketplace Demonstration

The marketplace demonstration is the primary evaluation scenario used in Chapter 6. It models an ancient marketplace populated by 30 NPCs belonging to three character types (vendors, guards, and visitors) along with environmental objects such as market stalls, benches, fountains, and guard posts. All behavioral variety emerges from our framework's memory-driven selection algorithm operating over JSON-defined behavior definitions.

Framework Manager

The concrete framework manager for this demo extends the wrapper layer's manager base class and implements the environmental query provider. The marketplace defines a single environmental condition representing the market's open or closed status. When our framework queries this condition through a callback, the provider returns the current status value from a market system component. The manager also handles market status change events: when the market transitions from closed to open, it triggers the framework's interruption mechanism for vendor NPCs, causing them to respond to the change through their configured interruption handlers.

Entity Types

The demo defines entity types that extend the wrapper layer's base classes. Framework entities (e.g. market stalls, benches, fountains, and guard areas) use the ambient entity base directly. NPCs extend the behavioral entity base class. Each character type defines which actions it can perform, corresponding to the actions available in their JSON configuration files.

Action Presentation

When our framework starts the execution of an action through a callback, the behavioral entity class translates this request into engine-specific operations (e.g. playing animations and issuing navigation commands). The specifics of how actions are presented visually are implementation decisions for the demo and are not determined by our framework. Our framework only requires that game engine systems eventually signal action completion.

5.1.4 Unity Editor Configuration

Setting up our framework in Unity's editor requires configuring several elements. Setup steps unrelated to our framework follow normal Unity workflows and are not covered in this section.

Configuration data asset

The framework configuration asset is created through Unity's asset creation menu. Figure 5.1 shows how the data asset contains the paths to all four JSON configuration files, the entity configuration folder, the log path, the log level, and the batch size. Developers can create multiple configuration assets for different scenarios in their games or testing configurations.

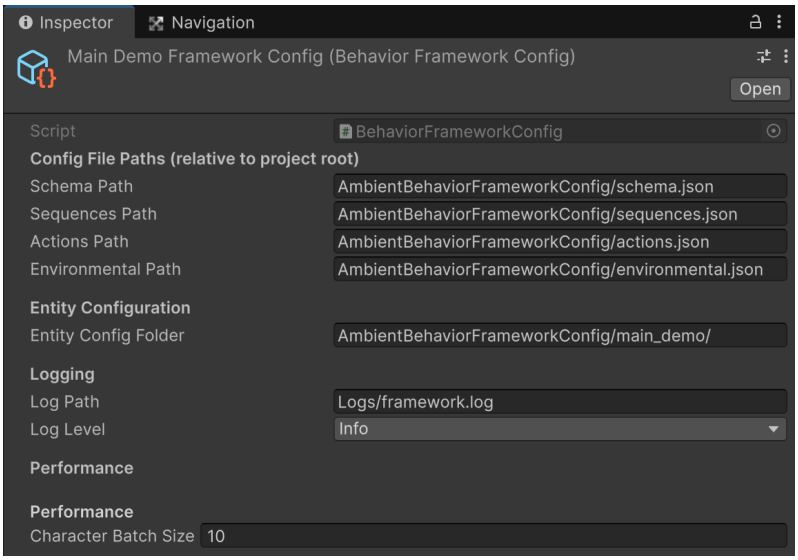

Figure 5.1: Configuration data asset for our main demonstration in Unity.

Framework Manager

A single game object holds the framework manager component. Figure 5.2 shows the editor's "Inspector" panel. Developers assign here the framework configuration data asset, which contains all JSON file paths and runtime parameters such as the batch size for the update

cycle of our framework. Developers also assign references to any game-specific systems that provide environmental conditions. When the game starts, the manager initializes our framework.

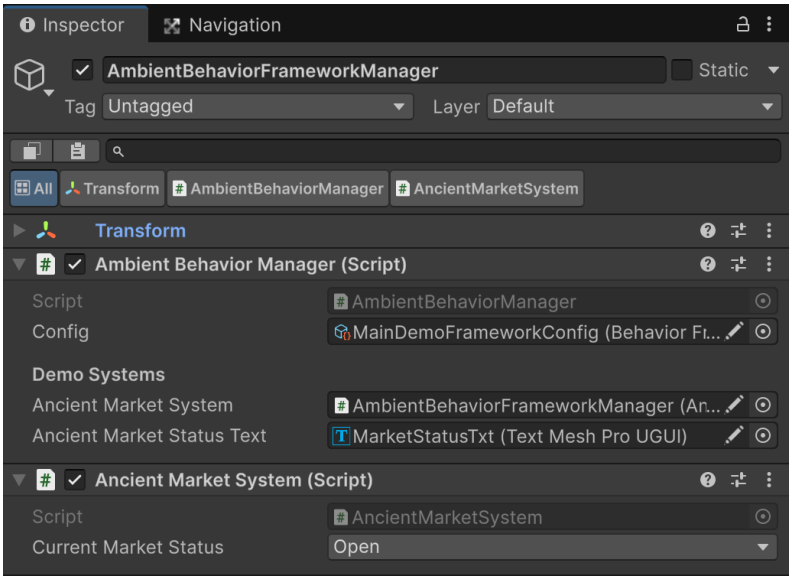

Figure 5.2: Framework manager game object configuration in Unity.

NPC Entities

Each NPC in the scene is a game object with a behavioral entity class component attached. Figure 5.3 shows the fields that developers must configure including the relative path to the entity's JSON file and a toggle for automatic registration in our framework.

Framework Entities

Non-behavioral objects such as market stalls and benches are configured as game objects with a framework entity class component. Figure 5.4 shows how each framework entity component specifies its JSON configuration path, the same field used by NPC entities. These game objects also have demo-specific components for managing spatial interactions with NPCs.

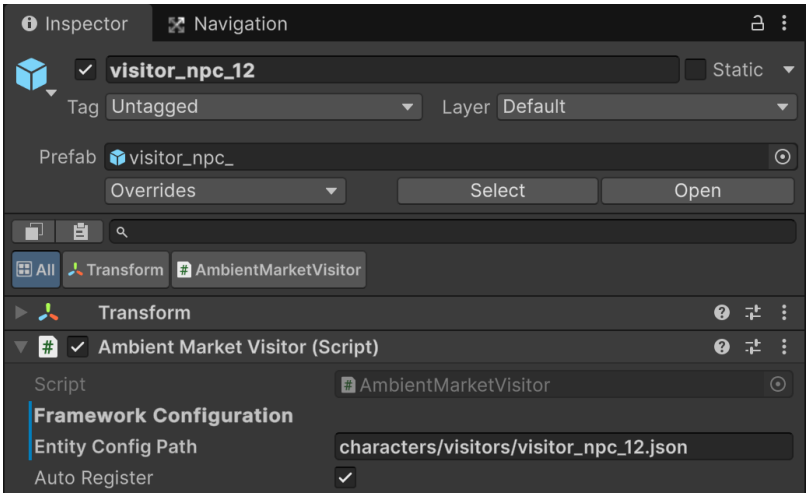

Figure 5.3: Example of an NPC game object configuration in Unity.

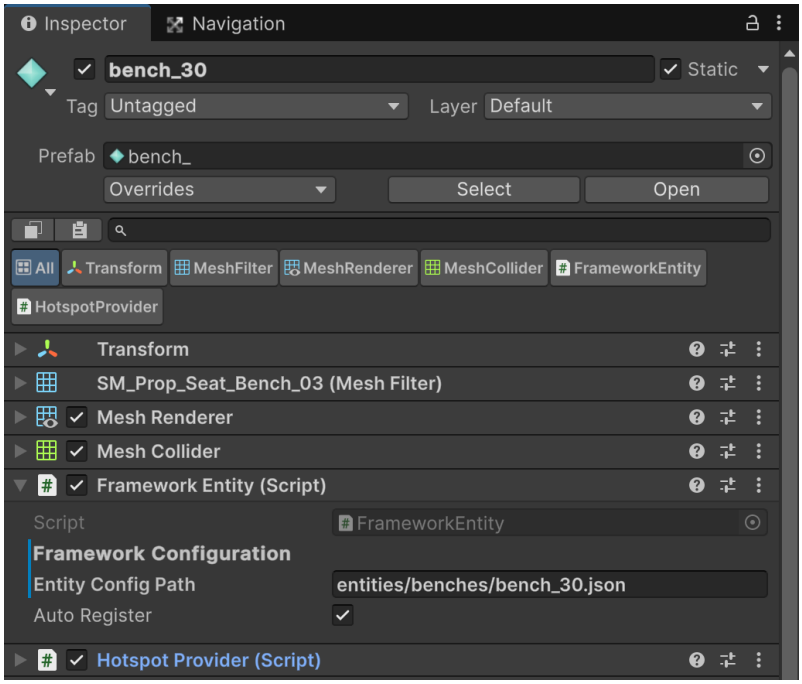

Figure 5.4: Example of a framework entity configuration in Unity.

5.2 Unreal Engine Integration

5.2.1 Engine Overview

Unreal Engine is a widely used game engine developed by Epic Games. Like Unity, it supports development across Windows, macOS, consoles, and mobile platforms. Unreal uses C++ as its scripting language and provides an additional visual scripting system called Blueprints. Unlike Unity, where gameplay code runs within a runtime that manages memory automatically, Unreal compiles gameplay code directly to native machine code. This means that there is no garbage collector that relocates objects in memory, which has implications for how our framework integration manages entity references (described in Section 5.2.2).

Unreal organizes games using an architecture similar to Unity's. Actors are objects placed in a level (analogous to Unity's game objects), and components attached to actors provide functionality such as rendering, physics, and custom behavior (just like Unity's component system). Components respond to engine lifecycle events through methods that mirror Unity's: `BeginPlay()` is called once at the beginning of the game, `Tick()` is called every frame, and `EndPlay()` is called when the actor is removed from the game.

Unreal's gameplay execution uses the same single-threaded model as Unity: all actor and component lifecycle methods execute on a single thread within each frame. As with Unity, the engine does not guarantee a specific ordering for how actors are processed during each frame. For NPC movement, Unreal provides a navigation system based on navigation meshes, similar to Unity's approach. NPC actors are assigned an AI controller component that handles pathfinding requests. For animation, Unreal uses a visual state machine system (called Animation Blueprints) that drives animation playback based on variables exposed from C++ code, similar to Unity's state machine system.

Unreal uses centimeters as its unit of measurement for positions and distances, in contrast to Unity's meters. This means that distance values in the framework's JSON configuration files must be scaled by a factor of 100 relative to values used in Unity configurations. For example, a distance precondition of 3 units in a Unity configuration becomes 300 in the equivalent Unreal Engine configuration. Our framework's distance calculations use Manhattan distance (as described in Section 4.4.2), which is coordinate-system agnostic.

5.2.2 Wrapper Layer Architecture

The Unreal wrapper layer implements the same coordination interface as the Unity wrapper and follows the same architectural pattern: A framework manager, entity base classes, a handle tracker, and a configuration asset.

Because Unreal gameplay code is compiled C++, there is no type conversion boundary when calling our framework's C API functions. Instead of declaring each function through a language interoperability mechanism as Unity does, the Unreal wrapper loads the DLL at runtime and resolves each exported function by name.

Entity handle management is also simpler than in Unity. Because Unreal does not use a garbage collector that relocates objects, the wrapper can use each entity's owning actor

pointer directly as the framework entity handle, eliminating the stable-reference allocation that the Unity wrapper requires.

One practical challenge specific to Unreal is the lifecycle event ordering when a level loads. Entity components may attempt to register with the framework manager before the manager's own initialization has completed. The wrapper addresses this through retroactive registration: After the manager finishes initializing, it scans the level for all entity components that should be registered but are not yet, and registers them. This ensures that all entities are registered regardless of initialization ordering.

5.2.3 Dance Club Demonstration

The Unreal Engine demonstration is designed to validate the engine-agnostic architecture of our framework. The scenario models a dance club environment with 10 NPCs: 5 dancers performing a simple looping dance sequence, and 5 visitors following the same main sequence structure as visitors in the ancient marketplace scenario but adapted to the dance club setting; browsing a bar counter instead of market stalls, walking around, sitting on couches, and drinking from tables.

The dance club scenario uses a simplified version of the ancient marketplace configuration files, with some actions removed and sequences updated for the new setting. The only parameter requiring adjustment between engines was the distance threshold in some action preconditions, which was converted from meters (Unity) to centimeters (Unreal Engine). Both engines use the same compiled DLL without modification. Behavioral logic and configuration file formats are identical, only the engine-specific wrapper layers and unit-dependent distance values differ.

The concrete manager for this demo follows the same pattern as the Unity implementation: It extends the wrapper's manager base class, implements the environmental query provider, and handles the retroactive entity registration described in Section 5.2.2. Entity types extend the wrapper's base classes and connect action requests from the framework to the engine's animation and navigation systems.

5.2.4 Unreal Engine Editor Configuration

The setup of our framework in Unreal Engine mirrors the Unity configuration. As with Unity, engine-standard setup such as navigation mesh generation and animation blueprint creation follows normal Unreal Engine workflows.

Configuration Data Asset

The framework configuration asset is created through Unreal's content browser. Figure 5.5 shows how the data asset contains the paths to all four JSON configuration files, the entity configuration folder, the log path, the log level, and the batch size. Developers can create multiple configuration assets for different scenarios in their games or for testing configurations.

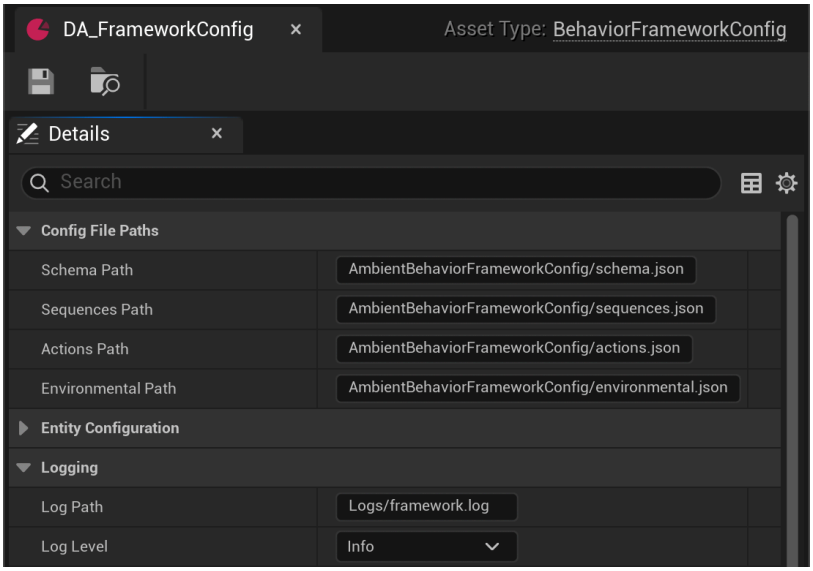

Figure 5.5: Configuration data asset for our Unreal Engine demonstration.

Framework Manager

A single actor in the level holds the concrete framework manager component. Figure 5.6 shows the editor's "Detail" panel. Developers assign here the framework configuration data asset, which contains all JSON file paths and runtime parameters such as the batch size for the update cycle of our framework. Developers also assign references to any game-specific systems that provide environmental conditions. When the game starts, the manager initializes our framework.

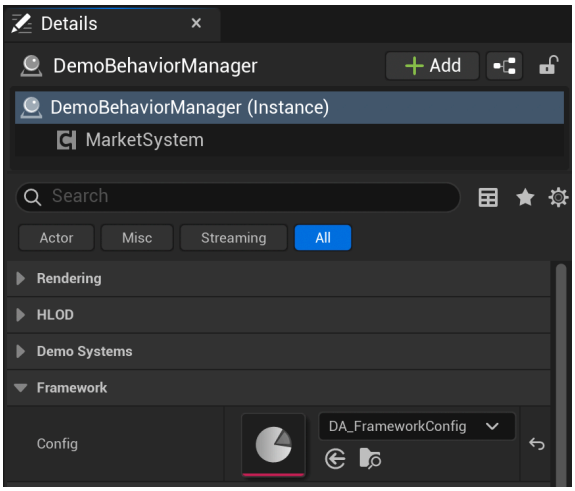

Figure 5.6: Framework manager game object configuration in Unreal Engine.

NPC Entities

Each NPC in the level is an actor with a behavioral entity class component attached. Figure 5.7 shows the fields that developers must configure including the relative path to the entity's JSON file and a toggle for automatic registration in our framework.

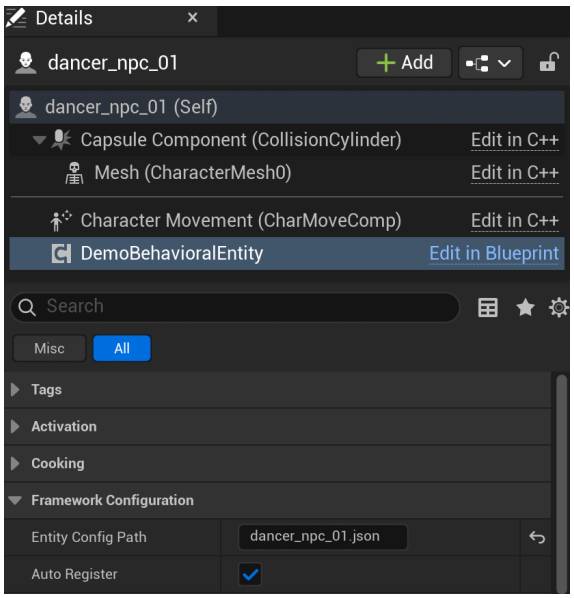

Figure 5.7: Example of an NPC game object configuration in Unreal Engine.

Framework Entities

Non-behavioral objects such as market stalls and benches are configured as actors with an ambient entity class component. Figure 5.8 shows how they specify their JSON configuration path just as NPC entities. These actors also have demo-specific components for managing spatial interactions with NPCs.

5.3 Integration Comparison

The two integrations implement the same framework coordination interface and follow the same architectural pattern (reusable wrapper layer with abstract base classes extended by demo-specific concrete classes), but they differ in several engine-specific details.

The most significant architectural difference is the language bridge. Unity's C# environment requires a type conversion layer at the DLL boundary: Every callback invocation crosses this boundary, incurring conversion overhead and requiring the wrapper to maintain stable entity references that survive garbage collection. In Unreal, DLL functions are called directly from native C++ with no type conversion boundary. This difference is reflected in the performance measurements in Chapter 6, where the engine-boundary overhead (position

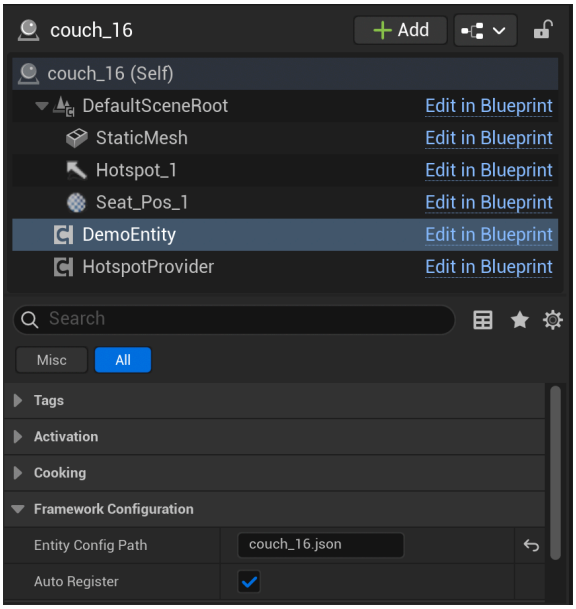

Figure 5.8: Example of a framework entity configuration in Unreal Engine.

synchronization and action initiation callbacks) constitutes the dominant per-frame cost in Unity.

Despite these differences, the integration effort for Unreal was lower than for Unity. The Unity integration was developed first and required designing the wrapper layer architecture, the entity class hierarchy, the handle tracking mechanism, and the action presentation approach. The Unreal integration followed the same architectural pattern, so the primary effort was translating existing design decisions into Unreal's conventions rather than making new ones. The wrapper layer classes have intentionally parallel structures: Each Unity class has a direct Unreal counterpart with the same responsibilities and extension points.

The elements that remained unchanged across both integrations are the framework DLL binary itself, and the JSON configuration files (with distance values adjusted for the unit system difference). The integration-specific work in each engine is limited to the wrapper layer (which implements the coordination interface) and the demo-specific layer (which translates framework action requests into engine-native animation and navigation). This separation validates the engine-agnostic architecture described in Section 4.1: decision-making for ambient NPCs is contained within the shared DLL, and each engine provides only a thin coordination layer connecting our framework to engine-specific systems.

6. Evaluation

In this chapter we present the evaluation of our framework against the five research objectives defined in Section 1.3. Three targeted demonstrations address some of these objectives. Table 6.1 summarizes the mapping between research objectives and evaluation methods. The ancient marketplace demonstration (Section 6.1), implemented in Unity, provides qualitative evidence for the core behavioral contributions of our framework (evaluating Objectives 1–3). The dance club demonstration (Section 6.2), implemented in Unreal Engine, validates that the decision-making components of our framework operate independently of game engine systems (Objective 4). The empirical evaluation, (Section 6.3) provides quantitative evidence for the computational efficiency of our implementation in Unity, validating Objective 5.

Research Objective	**Evaluation Method**	**Section**
1. Behavioral variety	Marketplace demo (qualitative)	6.1
2. Context-aware selection	Marketplace demo (qualitative)	6.1
3. Behavioral continuity	Marketplace demo (qualitative)	6.1
4. Engine-agnostic design	Unreal Engine demo (qualitative)	6.2
5. Computational efficiency	Performance testing (quantitative)	6.3

Table 6.1: Mapping of research objectives to evaluation methods.

6.1 Behavioral Demonstration

Our primary demonstration, shown in Figure 6.1 illustrates how the framework produces behavioral variety among a large number of NPCs that have identical behavior definitions. The scenario models an ancient marketplace implemented in Unity populated by 30 NPCs across three character types: 7 vendors, 5 guards, and 18 visitors. Each character type is defined through JSON configuration files specifying behavior sequences, actions, preconditions, and state properties. All NPCs of the same type share identical configuration files; any observed behavioral variation therefore emerges solely from the memory-driven selection algorithm rather than from per-character scripting.

The marketplace includes an environmental condition representing whether the market is open, which determines the behavioral paths available to each NPC type. When the market is open, vendors sell goods from their stalls (standing or sitting depending on their energy state), visitors walk between stalls to browse merchandise or rest or socialize when tired, and guards patrol designated areas. When the market closes, preconditions requiring an open market evaluate to false, causing NPCs to follow alternative behavioral paths: Vendors leave their stalls and behave like visitors, guards continue walking but no longer perform guard duty, and visitors can no longer browse stalls and instead choose among resting, socializing, or drinking. When the market reopens, vendors receive an interruption event that

Figure 6.1: Ancient marketplace demo in Unity.

triggers a "return to stall" sequence, restoring their selling routine. These context-dependent behavioral changes are implemented through preconditions and interruptions while using a single behavior definition for each character type.

The memory-driven selection algorithm produces behavioral variety at multiple levels of the sequence structure. At the top level, visitors choose among visiting a stall, resting, socializing, or drinking based on which transitions they have used least recently. Within nested sequences, additional decision points create further variation. For example, a visitor browsing a stall can select among different "browsing" actions, while a resting visitor can select among different idle behaviors. Entity selection adds another source of variation: When multiple stalls or benches satisfy an action's preconditions (e.g., available capacity and proximity to an NPC), the memory-driven algorithm selects among valid targets using action memory records. Over extended observation, NPCs sharing identical definitions follow distinct behavioral paths rather than repeating the same action patterns.

The demonstration also exercises the interruption and fallback mechanisms of the framework. A player-triggered shout event interrupts all NPCs within a certain distance of the player and causes them to execute a "surprised reaction" sequence. After the interruption sequence completes, NPCs with resumable actions return to their previous activities with preserved context (e.g., a visitor resumes browsing the same stall). When an action cannot be executed (e.g., all benches are occupied), the fallback mechanism activates a "walk around" sequence, ensuring that NPCs remain active until conditions change. These behaviors illustrate how the framework supports dynamic and varied NPC behavior without requiring per-character scripting or computationally expensive planning.

This behavioral assessment is qualitative and based only on personal observations when running the demonstration. Formal statistical analysis of behavioral variety is an important direction for future work.

6.2 Cross-Engine Validation

One of the goals of the framework is to allow the same behavioral logic to work across multiple game engines. To evaluate the engine-agnostic design of the framework, we deployed it in both Unity and Unreal Engine. The behavioral evaluation described in the previous section uses the ancient marketplace scenario in Unity. To validate that the same decision-making framework can work across different engines, we implemented a second scenario –a dance club environment in Unreal Engine (shown in Figure 6.2). This scenario contains 10 NPCs across two character types: Five dancers performing a simple looping dance sequence, and five visitors following the same behavior structure as the visitor NPCs in the ancient marketplace scenario. Visitors in this scenario browse a bar counter instead of market stalls and can also walk, sit on couches, or drink from tables. The Unreal Engine scenario uses a simplified version of the same behavioral configuration used in the ancient marketplace scenario, with some actions removed and sequences adjusted for the different environment.

Figure 6.2: Dance club demo in Unreal Engine.

Both scenarios use the same compiled DLL without modification. The behavioral logic remains identical across engines; only the engine-specific integration layer and unit-dependent parameters required adjustment. In particular, the distance thresholds used in entity preconditions were converted from meters (Unity) to centimeters (Unreal Engine) to match the coordinate systems of the engines.

These results demonstrate that the framework's behavioral decision algorithms can be reused across different game engines without modifying the framework implementation, needing only an engine-specific wrapper that connects engine systems to the framework's C API.

6.3 Performance Evaluation

6.3.1 Evaluation Methodology

As discussed in Section 1.5.1, real-time games operate under strict per-frame time budgets where all game subsystems, including rendering, physics, animation, audio, and AI, compete for processing time. The analysis of production AI budget practices in Section 2.3.3 indicates that commercial games typically allocate 5–10% of the total frame budget to the AI subsystem as a whole. This allocation covers the full range of AI responsibilities in a game, which can include combat reasoning, pathfinding decisions, perception, and ambient NPC behavior. Published examples reflect this combined scope: Based on the timing data reported by Plch et al., the complete AI system in *Kingdom Come: Deliverance* consumed approximately 6–12% of the frame budget depending on target frame rate [44, 45], and *Hitman Absolution* relies on time-slicing techniques to distribute AI computation for up to 300 NPCs across frames [16].

No published benchmarks or standardized protocols exist for evaluating the ambient behavioral decision-making component of a game's AI workload in isolation. Crowd simulation frameworks such as Menge [13] provide benchmarking suites for navigation and steering, but these address spatial movement rather than behavioral decision-making. Our framework addresses only ambient behavioral decision-making: Selecting what an NPC should do next based on its current context and behavioral history. It does not handle pathfinding, combat reasoning, perception, or other AI responsibilities. Because this is only one component of a game's total AI workload, a framework serving this purpose should consume substantially less than the full AI allocation to leave room for other AI subsystems. We therefore adopt 10% of the frame budget as a conservative upper-bound target, recognizing that this represents the entire AI allocation rather than just the portion for ambient behavior. At 60 FPS (16.67 ms per frame), 10% corresponds to approximately 1.67 ms; at 30 FPS (33.33 ms per frame), approximately 3.33 ms.

Framework cost is measured in milliseconds per frame and expressed as a percentage of total frame time, following the approach of expressing per-frame timing as a budget fraction used in the evaluation by Plch et al. [45]. Milliseconds per frame is the recommended metric for game performance analysis because FPS is non-linear [28, 52]. Per-frame timing is summarized using percentile statistics (P50 and P99) rather than simple averages, because percentile metrics characterize tail behavior more reliably than minimum and maximum values, which can be skewed by isolated garbage collection or scheduling events [17]. Performance is measured across four NPC population sizes (50, 100, 150, and 200), each tested five times to quantify run-to-run variability from non-deterministic factors such as garbage collection timing and operating system scheduling [24].

6.3.2 Test Protocol

All tests were performed on a desktop PC with an AMD Ryzen 9 7900X processor, an NVIDIA GeForce RTX 2070 graphics card, and 32 GB of RAM, running Windows. The demonstration scene ran in the Unity editor with the behavioral logic executing within a

C++ DLL invoked from Unity each frame.

A custom Unity script measures two values each frame: the total simulation time and the framework update time. The framework update measurement wraps the entire DLL update call, capturing both the iteration over all registered NPCs and the behavioral processing performed on active entities. A framework-side log records the number of NPCs that receive behavioral updates each frame, providing workload context for interpreting timing variation.

Each test begins with a 30-second warmup period to allow the simulation to reach a behaviorally representative state, followed by a 60-second measurement window. The camera remains stationary throughout all tests to eliminate rendering variability. Four NPC populations are tested (50, 100, 150, and 200), all using the batch size parameter set to -1, which represents the worst-case scheduling scenario where all NPCs are iterated every frame (as described in Section 4.4.1). Under this configuration, frames in which multiple NPCs happen to complete their current actions simultaneously will incur disproportionately higher processing costs, producing occasional spikes in per-frame timing. Each configuration is tested five times under identical conditions, yielding 20 timing runs. From each run, the average framework time, P50, and P99 framework time, average frame time, and average FPS are computed. Results present the mean and standard deviation across the five runs for each configuration.

Five additional runs at the 200-NPC configuration use the Tracy profiler to decompose per-function execution costs within the C++ DLL. These runs are performed separately from the timing measurements to avoid instrumentation overhead. Per-function mean time per call values are reported as the mean with standard deviation across the five profiling runs.

6.3.3 Timing Results

NPCs	Avg FPS	Avg Fwk Time (ms)	P50 Time (ms)	P99 Time (ms)	Fwk Cost (% Frame Budget)
50	152.2 (1.3)	0.126 (0.002)	0.102 (0.001)	0.718 (0.016)	**1.92 (0.02)**
100	131.6 (0.9)	0.186 (0.003)	0.134 (0.002)	0.863 (0.008)	**2.45 (0.03)**
150	121.0 (0.4)	0.245 (0.003)	0.167 (0.002)	1.002 (0.012)	**2.96 (0.03)**
200	112.0 (0.5)	0.305 (0.003)	0.203 (0.002)	1.173 (0.031)	**3.41 (0.02)**

Table 6.2: Framework timing performance. All tests process all NPCs each frame. Each cell reports the mean across five runs, with standard deviation in parentheses. Fwk = Framework.

Table 6.2 presents the timing results across all four NPC configurations. The framework budget percentage increases from 1.92% at 50 NPCs to 3.41% at 200 NPCs. Quadrupling the NPC population produces only a 1.8x increase in budget consumption, suggesting sub-linear scaling of the decision-making workload. This pattern is consistent with the framework's workload characteristics. Action durations are measured in seconds (typically 3 to 10 seconds in our configurations), while the simulation runs at frame rates exceeding 100 FPS. Only a small fraction of NPCs therefore require a behavioral decision on any given frame, and the fixed per-frame cost of iterating over all registered NPCs is amortized across larger

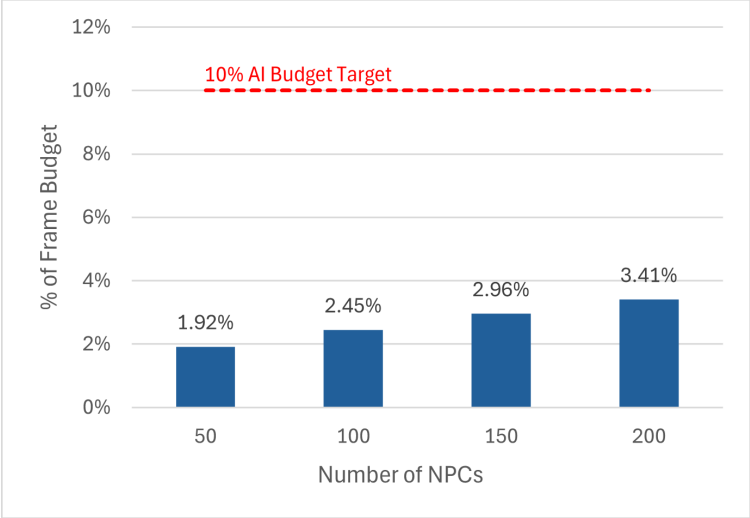

Figure 6.3: Framework overhead as a percentage of frame budget across NPC populations. All configurations remain well below the 10% target threshold.

populations. As shown in Figure 6.3, all configurations remain well below the 10% budget target, with the highest measured cost consuming approximately one-third of the allocation. These results indicate that our framework could support substantially larger ambient populations before approaching typical AI frame budget limits.

The P50 framework time is substantially lower than the average at all configurations (0.203 ms vs. 0.305 ms at 200 NPCs), indicating a right-skewed distribution where most frames involve minimal processing. Because all tests use a batch size of -1 (all NPCs iterated every frame), the higher-cost frames correspond to moments when multiple NPCs simultaneously complete their current actions and require new behavioral decisions on the same frame. Figure 6.4 shows this spread increasing with population size as the likelihood of simultaneous action completions grows. Even at P99, the 200-NPC configuration consumes only 1.173 ms, which is below the 1.67 ms that would represent 10% of a 60 FPS frame budget. The batch size parameter described in Section 4.4.1 provides a mechanism to distribute these processing spikes across multiple frames by limiting the number of NPCs iterated per frame, which would reduce P99 values at the cost of slightly increased latency for NPC behavior.

For context, the published AI systems described in Section 6.3.1 report 6–12% frame budget consumption for complete AI subsystems with broader responsibilities than ambient behavioral decision-making [16, 45]. A direct comparison of budget percentages between these systems and our framework is not appropriate given the difference in scope; rather, these published figures establish the total AI budget within which an ambient behavioral system must operate. The fact that our framework consumes at most 3.41% of the frame budget for ambient behavioral decision-making alone confirms that it would fit within a production AI budget alongside other AI components, without requiring time-sliced updates or level-of-detail scaling.

Run-to-run variability is minimal across all configurations. The measured framework budget percentage varies by at most 0.03 percentage points across five runs, indicating consistent and reproducible performance. The approximately 40 FPS reduction observed

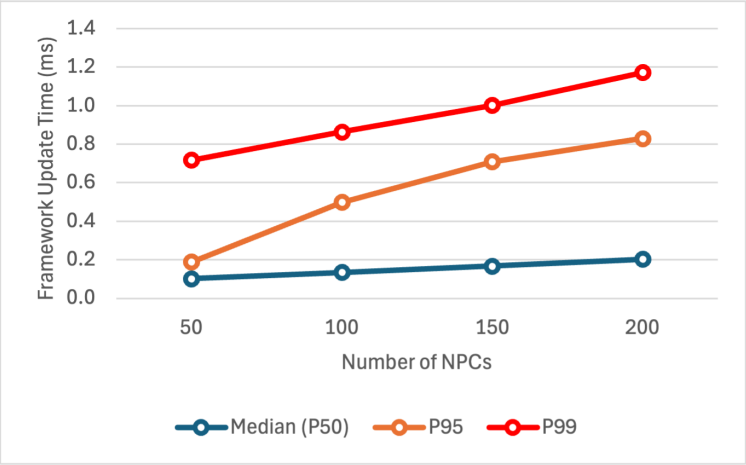

Figure 6.4: Framework update time distribution across NPC populations.

when increasing from 50 to 200 NPCs reflects the increased rendering and engine workload associated with simulating a larger number of characters, rather than framework overhead. Because framework cost is expressed as a percentage of actual frame time rather than a fixed budget target, the reported percentages remain valid across configurations regardless of this FPS variation.

6.3.4 Variable Workload Analysis

To provide further context for the scaling behavior observed in the previous section, we analyze how often NPCs actually require behavioral decisions. Our framework performs two types of work on each frame: A fixed cost proportional to the total NPC population (iterating and checking each entity) and a variable cost proportional to the number of entities that require behavioral decisions. Table 6.3 reports the average number of NPCs that received behavioral updates per frame for each configuration, contextualizing the per-frame timing variation observed in Table 6.2.

NPCs Registered	Avg Updated / Frame	Max Updated
50	0.2 (0.0)	3–4
100	0.4 (0.0)	5–7
150	0.6 (0.0)	10–20
200	0.8 (0.0)	19–36

Table 6.3: Frequency of NPC behavioral updates per frame. Avg Updated / Frame reports the mean across five runs, with standard deviation in parentheses. Max Updated shows the range of per-run maximum values.

Fewer than one NPC requires a behavioral decision on any given frame across all configurations. At 200 NPCs, the average is 0.8 updates per frame while iterating over all registered entities. This is expected: Action durations are defined in seconds, and at frame rates

above 100 FPS, the probability that any individual NPC completes its current action on a given frame is low. The maximum values show that occasional frames do require multiple simultaneous decisions (up to 36 at 200 NPCs), which explains the right-skewed per-frame distribution discussed in Section 6.3.3.

6.3.5 Internal Performance Breakdown

While the timing results demonstrate that the framework is inexpensive in aggregate, they do not indicate where time is spent internally. To examine this, we collected function-level profiling data using the Tracy profiler across five dedicated 200-NPC runs. These runs were performed separately from the timing experiments to avoid instrumentation overhead affecting the reported measurements. Table 6.4 summarizes the internal cost distribution across nine representative functions. Figure 6.5 groups these functions into three architectural categories: Decision Core, Update Loop, and Engine Boundary.

Function	**Calls**	**Total (ms)**	**MTPC (μs)**
SelectTransitionNodeId	2,665 (67)	9.34 (0.32)	3.50 (0.04)
SelectActionEntityId	857 (15)	3.31 (0.01)	3.86 (0.07)
EvaluatePreconditions	37,799 (860)	47.79 (1.45)	1.26 (0.02)
BehaviorFramework::Update	9,427 (74)	33.28 (0.43)	3.53 (0.05)
UpdateBehavioralEntities	9,426 (74)	1,790 (15.81)	189.88 (1.28)
ExecuteSequenceStep	7,651 (202)	38.65 (1.36)	5.05 (0.06)
GetActionTargetEntity	857 (15)	34.22 (0.90)	39.91 (0.38)
UpdateEntityPosition	746 (7)	175.01 (4.04)	234.46 (3.99)
InitiateActionExecution	2,285 (47)	490.51 (17.52)	214.60 (3.47)

Table 6.4: Internal performance breakdown at 200 NPCs. Each cell reports the mean across five profiling runs, with standard deviation in parentheses. MTPC = mean time per call.

The results show that the behavioral decision-making operations of our framework, grouped under the Decision Core category, are computationally inexpensive, operating at 1–4 µs per decision. The core operations (transition selection, entity selection, and precondition evaluation) contribute only a small fraction of the total runtime cost. In contrast, the largest cost arises at the engine boundary, particularly for position synchronization and action initiation. The `UpdateEntityPosition` and `InitiateActionExecution` functions exceed 200 µs per call, reflecting the overhead of data marshalling and engine-side operations rather than behavioral reasoning.

These costs are inherent to any architecture that separates decision-making logic from game engine systems. The majority of time is spent in operations at the interface between the framework and the game engine, which would be required regardless of the decision-making method used. Consequently, future optimization opportunities lie primarily in reducing engine-boundary overhead (for example, through batching updates or reducing interop calls) rather than in modifying the behavioral decision-making algorithm itself.

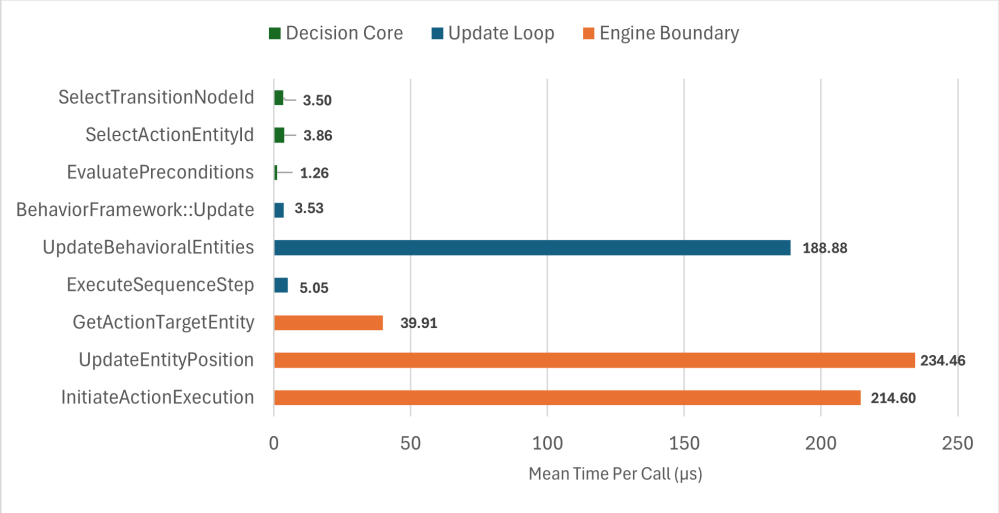

Figure 6.5: Mean time per call (MTPC) for key framework functions grouped by architectural category.

6.4 Evaluation Summary

The three evaluation approaches presented in this chapter provide evidence for each of the five research objectives defined in Section 1.3. The marketplace demonstration (Section 6.1) shows that the memory-driven selection algorithm produces behavioral variety across NPCs sharing identical definitions (Objective 1), that the unified condition evaluation mechanism (state operations) enables context-appropriate responses to changes in the game environment (Objective 2), and that the integrated resilience mechanisms maintain behavioral continuity under unexpected game conditions (Objective 3).

The cross-engine demonstration (Section 6.2) validates that the DLL-based architecture supports engine-agnostic operation, with the same behavioral logic and configuration producing equivalent behaviors in both Unity and Unreal Engine (Objective 4).

The performance evaluation (Section 6.3) demonstrates that the framework operates well within the 10% frame budget typically allocated to the complete AI subsystem in commercial games, peaking at 3.41% with 200 NPCs under worst-case scheduling without requiring time-sliced updates or level-of-detail scaling. Internal profiling shows that behavioral decision-making costs only 1–4 µs per decision, with the dominant runtime costs arising from engine-boundary communication rather than the selection algorithm itself (Objective 5). These profiling results indicate that the decision-making algorithm contributes a small fraction of per-frame cost at the tested scales, and that operations at the engine boundary would likely be the first constraint as populations grow.

Several limitations of this evaluation should be acknowledged. The behavioral assessment in Section 6.1 is qualitative, relying on observable properties rather than quantitative metrics such as action distribution histograms or repetition rates. A formal behavioral analysis with statistical measures of variety and context responsiveness as well as a user study group for qualitative assessment would strengthen the evidence for Objectives 1–3, providing an important direction for future work.

The performance evaluation was conducted on a single hardware configuration and within the Unity editor rather than a standalone build, which may not fully reflect production deployment conditions. In addition, the cross-engine validation uses a scene with 10 NPCs, confirming functional correctness but not evaluating performance parity between engines.

Despite these limitations, the combined evidence indicates that the framework achieves its stated objectives within the scope of the demonstrations presented. The evaluation demonstrates that our framework produces varied and context-responsive NPC behavior while remaining computationally inexpensive and portable across game engines.

7. Conclusions

This chapter summarizes the contributions of our work, positions our framework among existing approaches to ambient NPC behavior, discusses its limitations, directions for future work, and concludes with final remarks on its broader applicability.

7.1 Summary of Contributions

This report presents a scalable framework for ambient NPC behavior that addresses the trade-off between behavioral variety and computational efficiency in large NPC populations. The framework was evaluated through qualitative behavioral demonstrations in two game engines and quantitative performance testing scaling from 50 to 200 NPCs. The following paragraphs summarize each contribution and the evidence supporting it.

Memory-Driven Action Selection Algorithm

Each NPC maintains a record of recent decisions and uses this history to guide future behavioral choices toward variety. When an NPC faces a choice between multiple valid options (transitions at decision points in a sequence, or several entities available for an action), the algorithm favors options the NPC has not tried recently (Section 3.1.5). The algorithm was implemented using bounded queues with configurable capacity limits per memory type (Section 4.2.3). The marketplace demonstration (Section 6.1) showed that NPCs sharing identical behavior definitions produced distinct behavioral patterns over time. Internal profiling confirmed that the per-decision cost of the selection algorithm is 1–4 µs (Section 6.3.5).

Unified State Operation Mechanism

A state operation specifies a target (the NPC, another entity, a game environmental condition, or the distance to another entity), a state property (e.g., energy level or occupied seats), an operation type (comparison or modification), and a value (Section 3.1.2). Whether the framework checks an NPC's energy, a bench's available capacity, or the current weather, it uses this same structure. When configured with a comparison operation, a state operation serves as a precondition that evaluates whether an action can execute or a transition can be taken. When configured with a modification operation, it serves as an effect that updates entity state at action start, completion, or interruption (Section 3.1.3). This unified mechanism operates identically across both the Unity marketplace demo and the Unreal Engine dance club demo (Section 6.2).

Integrated Behavioral Resilience Mechanisms

Three mechanisms work together to keep NPCs active when unexpected conditions arise. Interruption handling preserves the context of an interrupted action (the action, the sequence node, and the entity being used) and attempts resumption after the interrupting sequence completes (Section 3.3.4 and Section 3.3.5). Fallback sequences activate when a behavior sequence enters a failed state (for example, when no transitions from the current node have satisfied preconditions or when no entities satisfy an action's requirements), ensuring that NPCs transition to safe default behaviors rather than remaining without a valid behavioral path. A consecutive-failure safeguard halts behavioral processing if an NPC exceeds a configurable threshold of consecutive failures, preventing misconfigured NPC definitions from repeatedly consuming computational resources (Section 3.3.6). The marketplace demonstration exercised all three mechanisms: Player-triggered shout events interrupted NPCs who then resumed their previous activities with preserved context, resource contention (all benches occupied) activated fallback sequences, and the market open/close cycle triggered interruption events that redirected vendor NPCs to their selling routines.

Engine-Agnostic, Data-Driven Implementation

The framework is implemented as a C++ dynamic-link library exposing a public C API, with all behavior specifications defined in JSON configuration files which are loaded at runtime (Chapter 4). Game engines integrate with the framework by providing three callback functions (action execution, environmental condition queries, and entity position queries) and calling the public API for framework lifecycle, entity management, and runtime operations. The same compiled DLL and configuration files (with distance values adjusted for the coordinate system difference between Unity's meters and Unreal Engine's centimeters) produced equivalent behaviors in both engines, with each engine requiring only a wrapper layer to connect the framework's interface to engine-specific systems (Chapter 5).

7.2 Positioning Among Existing Solutions

Chapter 2 analyzes existing approaches to NPC behavior along several dimensions: Computational cost, behavioral variety, scalability, authoring complexity, and engine coupling. This section positions our framework among those approaches based on the architectural and algorithmic properties described throughout this report. This positioning is based on analysis of published information and the evaluation presented in Chapter 6, not on a controlled experiment directly comparing runtime performance or behavioral output across systems. A formal quantitative comparison remains an important direction for future work.

Planning-based algorithms such as GOAP and HTN (Section 2.1.3) generate varied and context-sensitive action sequences by searching through possible plans that satisfy character goals. This search produces behavioral diversity but incurs a per-decision computational cost that makes these algorithms impractical for the large ambient NPC populations that open-world games require. Our framework avoids planning search altogether, replacing it with a bounded memory lookup that operates at 1–4 µs per decision, a cost comparable to

reactive systems.

Utility-based systems (Section 2.1.2) produce dynamic, responsive behavior by evaluating scoring functions across possible actions and selecting the one with the highest utility. This approach naturally handles trade-offs between competing priorities and produces varied behavior, but it requires careful tuning of evaluation functions and weights for each action and character type. Our framework replaces numerical scoring with historical tracking: Rather than calculating which option is best, it tracks what each NPC has done recently and favors what it has not done. This produces varied selection among available options without requiring designers to define and tune scoring functions for each action.

Reactive systems such as FSMs and behavior trees (Section 2.1.1 and Section 2.1.2) scale well and have low per-decision cost, but they produce repetitive patterns because they evaluate options in a fixed order. Our framework uses a directed-graph structure similar to these reactive approaches but adds a memory-driven selection layer on top. The structural overhead of this layer is small (bounded memory containers and timestamp comparisons), preserving the efficiency of reactive methods while addressing their primary limitation.

Schedule systems (Section 2.1.4) create believable daily routines by assigning activities to time blocks, but they are inherently repetitive within those blocks. Our framework can incorporate time-based constraints through environmental condition preconditions (such as time of day or market open/closed status) while the memory system adds variety within those time-constrained behavioral paths.

CIVIL AI (Section 2.4.1) is the most directly comparable existing framework to our own, as it specifically targets ambient NPC behavior. It integrates utility-based decision making, HTN planning, smart objects, personality traits, and an influence map into a comprehensive system. Our framework is narrower in scope: it does not include personality traits, influence maps, or social relationships. However, it achieves behavioral variety through a single lightweight mechanism rather than multiple integrated algorithms, and its engine-agnostic architecture allows the same behavioral logic to operate across different game engines, while CIVIL AI is tightly coupled to Unity.

Two recent research approaches address behavioral variety for large NPC populations. Cardon and Jacopin accelerate GOAP planning using GPU parallelism, enabling thousands of NPC plans per frame, but their approach still relies on per-decision search and requires GPU resources that may compete with rendering. Bulitko et al. use evolutionary computation to generate collections of NPC behaviors offline and neural networks to select the most interesting ones, but their approach requires offline precomputation before gameplay. Our framework produces variety at runtime without search, GPU resources, or offline preprocessing.

In summary, our framework occupies a position between reactive systems and deliberative systems: it preserves the low per-decision cost and scalability of reactive approaches while producing the behavioral variety typically associated with more computationally expensive algorithms. The trade-off is that our framework does not optimize decisions (it does not select the best action for a given context, only one the NPC has not done recently) and does not model goals, utilities, or plans. For ambient NPCs performing routine background activities, this trade-off is appropriate because the primary requirement is variety and contextual plausibility rather than optimality.

7.3 Limitations and Future Work

This section discusses the limitations of our framework and its evaluation, along with directions for future work that could address them.

Evaluation Scope

The behavioral assessment in Section 6.1 is qualitative, relying on observable properties of NPC behavior rather than quantitative metrics such as action distribution entropy or repetition frequency. Without such metrics, no formal comparison with other NPC systems (for example, measuring behavioral variety and frame budget consumption against a utility system, a behavior tree, or CIVIL AI running equivalent scenarios) can be made. The positioning presented in Section 7.2 is therefore based on architectural and algorithmic analysis of published information rather than on controlled experiments comparing behavioral output or runtime performance. Developing quantitative behavioral metrics and applying them in a head-to-head comparison would provide evidence for the trade-offs described in this report.

A formal user study has also not been conducted. Evaluating the framework with game developers would assess the usability of the JSON authoring workflow, while evaluating the framework with players would determine whether the behavioral variety produced by the memory-driven selection algorithm is perceptible during gameplay.

All performance measurements were collected on a single high-end desktop (using an AMD Ryzen 9 7900X processor, an NVIDIA RTX 2070 graphics card, and 32 GB of RAM) running within the Unity editor rather than a standalone build. The editor introduces overhead that is absent in production builds, making these results a conservative estimate, but they do not represent the range of consumer hardware on which games run. The cross-engine validation in Unreal Engine confirmed functional correctness with 10 NPCs but did not evaluate performance parity between engines at scale. Future work should evaluate the framework on a broader range of hardware configurations, in standalone builds, and with performance comparisons across engines at equivalent NPC population sizes.

Framework Design Scope

All entity state managed by the framework is internal. Game engine systems cannot directly access the property values that the framework maintains for NPCs and other entities. If a game needs to use framework-managed state in other systems (for example, displaying an NPC's energy level in a user interface), the current architecture requires the game engine to maintain a parallel copy of that state. Providing a read-only query interface for entity state through the public C API would address this limitation without compromising the framework's encapsulation.

The memory system tracks only individual action history through three memory types: Transition memories, action memories, and interruption memories. There is no memory type for recording interactions with other characters or with the player. Some commercial games already implement this capability, notably *Red Dead Redemption 2* (Section 2.5.4) whose NPCs remember the player's appearance and past interactions. Adding new memory

types for interactions would require modifications to the selection algorithm to incorporate interaction history into decision-making, but the existing architecture of bounded memory record containers provides a foundation for this extension.

The framework does not model social relationships or coordinated group behavior. NPCs make decisions independently based on their individual memory, entity state, and environmental conditions. Group activities, relationship-influenced behavior, and crowd dynamics (Section 2.6.3) are outside the current scope. This was a deliberate scoping decision: the problem statement (Section 1.2) focuses on behavioral variety and environmental awareness as prerequisites for meaningful social behavior. Extending the framework with social simulation capabilities is a natural direction for future work.

Time Synchronization

The framework uses wall-clock (real-world) time internally, while game engines use game time that can be paused, slowed, or accelerated. Scenarios that break continuous time synchronization (game pausing, debugging, or engine hitches) cause temporary behavioral inconsistencies as the framework's internal timestamps diverge from the engine's simulation state. For example, if a game is paused for 30 seconds, the framework's memory records appear 30 seconds older than they should relative to in-game time, potentially affecting which options the selection algorithm favors. A production implementation would need a time synchronization protocol where the engine provides simulation time to the framework rather than the framework reading wall-clock time independently.

Configuration Authoring

JSON configuration files must currently be authored manually. For the demonstrations presented in this report (30 NPCs with three character types and approximately 30 actions), manual authoring was manageable. However, for projects with dozens of actions and sequences and hundreds of entity configurations, manual JSON editing could become error-prone and difficult to organize. The behavioral quality also depends on careful configuration design: Appropriate preconditions, well-structured sequences, and correct parameter values (such as distance thresholds). During development, an incorrect distance precondition value caused NPC behavior issues that initially appeared to be bugs in the memory-driven action selection algorithm, illustrating how configuration errors can mimic deeper problems. Visual tools for generating and validating configuration files would reduce this authoring burden and make the framework more accessible to game designers who may not be comfortable editing JSON files directly.

Threading and Scalability

The framework executes entirely on the game engine's main thread. Performance evaluation showed that this approach is sufficient for up to 200 NPCs while consuming less than 3.5% of the frame budget. For larger populations, partitioning entity updates across worker threads could reduce per-frame cost. The framework's architecture, where each NPC's decision-making depends only on its own memory, entity state queries, and environmental condi-

tions, is amenable to parallel processing, but the current implementation does not exploit this. The batch size parameter already supports distributing updates across multiple frames as an alternative to parallelization. Combining level-of-detail techniques (such as reducing memory capacity and update frequency for distant NPCs) with the existing batch processing mechanism could extend the effective population scale without requiring architectural changes.

7.4 Final Remarks

The central finding of this report is that lightweight memory-based decision algorithms can provide a practical alternative to planning and utility-based systems for controlling large populations of ambient NPCs. By tracking what each NPC has done recently and favoring what it has not done, the framework produces behavioral variety at a per-decision cost of 1–4 µs, comparable to the reactive systems it is designed to improve upon.

The data-driven, engine-agnostic architecture is not inherently limited to games. Any real-time simulation that requires varied, context-aware behavior for large populations of autonomous agents (such as training simulations, architectural walkthroughs, or virtual environments for research) could use the same DLL and configuration-based approach. The memory-driven selection principle itself, tracking recent choices to favor unexplored or least-recently-used options, is a general strategy for producing variety in any selection problem where exhaustive evaluation is too expensive.

The framework addresses a specific and practical challenge in game development: Making ambient NPCs behave with enough variety and contextual plausibility that players perceive a living world, while consuming few computational resources so that the game can support the hundreds of characters that modern open-world games require.

Bibliography

[1] AI and Games. *How Watch Dogs: Legion's 'Play as Anyone' Simulation Works*. Accessed: Mar. 25, 2026. Dec. 2020. URL: https://www.youtube.com/watch?v=SXn_c-HM0Vk.

[2] Hendrawan Armanto et al. "Improved Non-Player Character (NPC) Behavior Using Evolutionary Algorithm—A Systematic Review". In: *Entertainment Computing* 52 (Jan. 2025), p. 100875. ISSN: 18759521. DOI: 10.1016/j.entcom.2024.100875.

[3] Joseph Bates. "The Role of Emotion in Believable Agents". In: *Communications of the ACM* 37.7 (July 1994), pp. 122–125. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/176789.176803.

[4] Shakir Belle, Curtis Gittens, and T.C. Nicholas Graham. "Programming with Affect: How Behaviour Trees and a Lightweight Cognitive Architecture Enable the Development of Non-Player Characters with Emotions". In: *2019 IEEE Games, Entertainment, Media Conference (GEM)*. New Haven, CT, USA: IEEE, June 2019, pp. 1–8. DOI: 10.1109/GEM.2019.8811542.

[5] Matt Bertz. *The Elder Scrolls V: Skyrim Preview - The Technology Behind The Elder Scrolls V: Skyrim*. Game Informer. Accessed: Mar. 25, 2026. Nov. 2011. URL: https://gameinformer.com/games/the_elder_scrolls_v_skyrim/b/xbox360/archive/2011/01/17/the-technology-behind-elder-scrolls-v-skyrim.aspx.

[6] Daniel Brewer and Rez Graham. *Knowledge Is Power: An Overview of Knowledge Representation in Game AI*. Accessed: Mar. 25, 2026. 2020. URL: https://www.youtube.com/watch?v=Z6oZnDIgio4.

[7] Vadim Bulitko et al. "Towards Positively Surprising Non-Player Characters in Video Games". In: *Proceedings of the AIIDE Conference* 13.2 (Oct. 2017), pp. 34–40. DOI: 10.1609/aiide.v13i2.12981.

[8] Chris Callison-Burch et al. *Dungeons and Dragons as a Dialog Challenge for Artificial Intelligence*. 2022. DOI: 10.48550/ARXIV.2210.07109.

[9] Stephane Cardon and Eric Jacopin. "Binary GPU-Planning for Thousands of NPCs". In: *2020 IEEE Conference on Games (CoG)*. Osaka, Japan, Aug. 2020, pp. 678–681. DOI: 10.1109/CoG47356.2020.9231696.

[10] Martin Cerny et al. "Spice It Up! Enriching Open World NPC Simulation Using Constraint Satisfaction". In: *Proceedings of the AIIDE Conference* 10.1 (Oct. 2014), pp. 16–22. ISSN: 2334-0924, 2326-909X. DOI: 10.1609/aiide.v10i1.12715.

[11] Carle Cote. "Reactivity and Deliberation in Decision-Making Systems". In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 137–147. ISBN: 978-1-4665-6596-8.

[12] Francois Cournoyer and Antoine Fortier. *Massive Crowd on Assassin's Creed Unity: AI Recycling*. Accessed: Mar. 25, 2026. Jan. 2021. URL: https://www.youtube.com/watch?v=Rz2cNWVLncI.

[13] Sean Curtis, Andrew Best, and Dinesh Manocha. “Menge: A Modular Framework for Simulating Crowd Movement”. In: *Collective Dynamics* 1 (Mar. 2016), A1. DOI: 10.17815/CD.2016.1.

[14] Muyun Dai, Chun Yuan, and Xiaomei Nie. “Managing the Personality of NPCs with Your Interactions: A Game Design System Based on Large Language Models”. In: *HCI in Games*. Ed. by Xiaowen Fang. Vol. 14730. Cham: Springer Nature Switzerland, 2024, pp. 247–259. DOI: 10.1007/978-3-031-60692-2_17.

[15] Michael Dawe et al. “Behavior Selection Algorithms”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 47–60. ISBN: 978-1-4665-6596-8.

[16] Maurizio De Pascale and Mika Vehkala. *Creating the AI for the Living, Breathing World of Hitman: Absolution*. Accessed: Mar. 25, 2026. 2013. URL: https://gdcvault.com/play/1017802/Creating-the-AI-for-the.

[17] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 74–80. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2408776.2408794.

[18] Kevin Dill. “Structural Architecture - Common Tricks of the Trade”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 61–71. ISBN: 978-1-4665-6596-8.

[19] Christopher Dragert. *Watch Dogs: Legion | The Design and Tech Behind Play As Anyone*. Accessed: Mar. 25, 2026. Dec. 2020. URL: https://www.youtube.com/watch?v=mMFqK7dn5Wo.

[20] Epic Games. *Artificial Intelligence in Unreal Engine*. Accessed: Mar. 25, 2026. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/artificial-intelligence-in-unreal-engine.

[21] Wes Fenlon. *Cyberpunk 2077 Director Says Studio's Switch from REDengine to Unreal Engine 5 'Isn't Starting from Scratch'*. Accessed: Mar. 25, 2026. Sept. 2023. URL: https://www.pcgamer.com/cyberpunk-2077-director-says-studios-switch-from-redengine-to-unreal-engine-5-isnt-starting-from-scratch/.

[22] Game Maker's Toolkit. *The Genius AI Behind The Sims*. Accessed: Mar. 25, 2026. June 2023. URL: https://www.youtube.com/watch?v=9gf2MT-IOsg.

[23] GDC. *2026 State of the Game Industry*. Accessed: Mar. 25, 2026. 2026. URL: https://images.reg.techweb.com/Web/UBMTechweb/%7Bfbdfe6c4-e33f-458e-a8ac-96db55fda684%7D_541400_GDC26_PDF_SOTI_Report_Final.pdf.

[24] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*. Montreal Quebec Canada: ACM, Oct. 2007, pp. 57–76. DOI: 10.1145/1297027.1297033.

[25] Ran Gong et al. *MindAgent: Emergent Gaming Interaction*. Sept. 2023. DOI: 10.48550/arXiv.2309.09971. eprint: 2309.09971 (cs).

[26] David Graham. “Breathing Life into Your Background Characters”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 451–458. ISBN: 978-1-4665-6596-8.

[27] David "Rez" Graham. “An Introduction to Utility Theory”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 113–126. ISBN: 978-1-4665-6596-8.

[28] Jason Gregory. *Game Engine Architecture*. Third edition. Boca Raton London New York: CRC Press, Taylor & Francis Group, 2019. ISBN: 978-1-138-03545-4.

[29] Daniel Hilburn. “Simulating Behavior Trees. A Behavior Tree/Planner Hybrid Approach”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 99–111. ISBN: 978-1-4665-6596-8.

[30] Troy Humphreys. “Exploring HTN Planners through Example”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 149–167. ISBN: 978-1-4665-6596-8.

[31] Anja Johansson and Pierangelo Dell'Acqua. “Comparing Behavior Trees and Emotional Behavior Networks for NPCs”. In: *2012 17th International Conference on Computer Games (CGAMES)*. Louisville, KY, USA: IEEE, July 2012, pp. 253–260. DOI: 10.1109/CGames.2012.6314584.

[32] Yasemin Karaca, Djameleddine Derias, and Gözde Sarsar. “AI-Powered Procedural Content Generation: Enhancing NPC Behaviour for an Immersive Gaming Experience”. In: (Dec. 2023). DOI: 10.13140/RG.2.2.23719.52647.

[33] Pattie Maes. “Artificial Life Meets Entertainment: Lifelike Autonomous Agents”. In: *Communications of the ACM* 38.11 (Nov. 1995), pp. 108–114. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/219717.219808.

[34] Josh McCoy, Michael Mateas, and Noah Wardrip-Fruin. “Comme Il Faut: A System for Simulating Social Games Between Autonomous Characters”. In: *Digital Arts and Culture* (2009). Accessed: Mar. 25, 2026. URL: https://escholarship.org/uc/item/6x5933cw.

[35] Bill Merrill. “Building Utility Decisions into Your Existing Behavior Tree”. In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 127–136. ISBN: 978-1-4665-6596-8.

[36] Robert Meyer. *Tree's Company: Systemic AI Design in Just Cause 3*. Accessed: Mar. 25, 2026. Mar. 2019. URL: https://www.youtube.com/watch?v=SurYVTMINhg.

[37] Ian Millington. *AI for Games*. 3rd ed. Boca Raton, FL: CRC Press, Mar. 2019. ISBN: 978-1-351-05330-3. DOI: 10.1201/9781351053303.

[38] Shohei Miyashita et al. “Developing Game AI Agent Behaving like Human by Mixing Reinforcement Learning and Supervised Learning”. In: *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. Kanazawa, Japan: IEEE, June 2017, pp. 489–494. DOI: 10.1109/SNPD.2017.8022767.

[39] Gustavo Morais et al. “CST-Godot: Bridging the Gap Between Game Engines and Cognitive Agents”. In: *2022 21st Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. Natal, Brazil: IEEE, Oct. 2022, pp. 1–6. DOI: 10.1109/SBGAMES56371.2022.9961082.

[40] Ruslan Omirgaliyev, Damir Kenzhe, and Suienish Mirambekov. "Simulating Life: The Application of Generative Agents in Virtual Environments". In: *2024 IEEE AITU: Digital Generation*. Astana, Kazakhstan: IEEE, Apr. 2024, pp. 181–187. DOI: 10.1109/IEEECONF61558.2024.10585387.

[41] Jeff Orkin. *Applying Goal-Oriented Action Planning to Games*. AI Game Programming Wisdom 2 draft. Accessed: Mar. 25, 2026. 2003. URL: https://web.archive.org/web/20230912173044/https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf.

[42] Jeff Orkin. "Three States and a Plan: The A.I. of F.E.A.R." In: *Proceedings of the Game Developers Conference (GDC)*. Accessed: Mar. 25, 2026. 2006. URL: https://www.gamedevs.org/uploads/three-states-plan-ai-of-fear.pdf.

[43] Joon Sung Park et al. *Generative Agents: Interactive Simulacra of Human Behavior*. Aug. 2023. DOI: 10.48550/arXiv.2304.03442. eprint: 2304.03442 (cs).

[44] Tomas Plch. "Believable Decision Making in Large Scale Open World Games for Ambient Characters". Doctoral. Prague: Charles University, 2017.

[45] Tomas Plch et al. "An AI System for Large Open Virtual World". In: *Proceedings of the AIIDE Conference* 10.1 (Oct. 2014), pp. 44–51. DOI: 10.1609/aiide.v10i1.12705.

[46] Steve Rabin. *Game AI Pro 360: Guide to Character Behavior*. Game AI Pro 360. CRC Press, 2019. ISBN: 978-0-429-62180-2.

[47] Ben Sunshine-Hill. "Phenomenal AI Level-of-Detail Control with the LOD Trader". In: *Game AI Pro*. Vol. 1. CRC Press, Sept. 2013, pp. 185–200. ISBN: 978-1-4665-6596-8.

[48] Penelope Sweetser and Janet Wiles. "Combining Influence Maps and Cellular Automata for Reactive Game Agents". In: *Intelligent Data Engineering and Automated Learning - IDEAL 2005*. Ed. by David Hutchison et al. Vol. 3578. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 524–531. DOI: 10.1007/11508069_68.

[49] Penny Sweetser. "Large Language Models and Video Games: A Preliminary Scoping Review". In: *ACM Conversational User Interfaces 2024*. Luxembourg Luxembourg: ACM, July 2024, pp. 1–8. DOI: 10.1145/3640794.3665582.

[50] Tommy Thompson. *The AI of Hitman (2016)*. Accessed: Mar. 25, 2026. 2019. URL: https://www.gamedeveloper.com/design/the-ai-of-hitman-2016-.

[51] Bard Tree Limited. *Introduction to CIVIL-AI-SYSTEM*. Accessed: Mar. 25, 2026. Jan. 2026. URL: https://bardtreelimited.com/docs/guide-basic-intro-civil-ai-system.

[52] Unity Technologies. *Best Practices for Profiling Game Performance*. Accessed: Mar. 25, 2026. URL: https://unity.com/how-to/best-practices-for-profiling-game-performance.

[53] Valerio Velardo. *The Reality of Red Dead Redemption 2's AI (Part 1)*. Accessed: Mar. 25, 2026. Mar. 2019. URL: https://medium.com/the-sound-of-ai/the-reality-of-red-dead-redemption-2s-ai-part-1-c276e9da2763.

[54] Ming Yan et al. *LARP: Language-Agent Role Play for Open-World Games*. Dec. 2023. DOI: 10.48550/arXiv.2312.17653. eprint: 2312.17653 (cs).

[55] Shijie Zheng et al. "MemoryRepository for AI NPC". In: *IEEE Access* 12 (2024), pp. 62581–62596. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3393485.

A. Configuration File Specifications

This appendix describes the format and fields of each JSON configuration file used by our framework. The examples throughout the appendix are drawn from the ancient marketplace demo described in Chapter 5. Each section explains the purpose of each configuration file, provides an annotated example with field-by-field explanations, and documents the specifications that developers must follow when authoring these files. Readers are assumed to be familiar with the framework concepts introduced in Chapter 3 and the implementation details described in Chapter 4.

A.1 Configuration Files Dependencies

Our framework loads configuration files in a fixed order during initialization. This order exists because each file can reference identifiers specified by previously loaded files.The loading order is as follows. The **state schema** is loaded first, establishing the mappings between human-readable names and integer keys used for entity state properties, operation types, and interruption handlers. **Environmental conditions** are loaded second, defining the game conditions that can be referenced by state operations. **Actions** are loaded third, referencing state key names from the schema and condition names from the environmental conditions file. **Behavior sequences** are loaded fourth, referencing action identifiers from previously loaded files. **Entity** configuration files are loaded at runtime when each entity is registered with the framework, referencing action identifiers, sequence identifiers, and state key names that must already have been defined.

Our framework checks that all identifiers referenced in a configuration file correspond to identifiers that have already been registered from previously loaded files. If a referenced identifier does not exist (for example, an action referencing a state key name that was not defined in the state key schema file), the framework logs an error and rejects the file. For this reason, developers must ensure that all referenced identifiers exist in the appropriate files and that files are loaded in the correct order during framework initialization. Our C++ DLL implementation of the framework already handles the correct loading order of the configuration files. Developers using our DLL as a starting point are just required to provide the appropriate wrapper layer with their game engine of choice following the specifications described in Chapter 5.

A.2 State Schema Format

The state schema file defines the vocabulary that all other configuration files use. It establishes mappings between human-readable names and the integer keys that the framework uses internally at runtime.

Listing A.1: Example of a schema configuration file.

```
{
  "entity_states": [
    { "name":  "CAPACITY", "key":  0},
    { "name":  "ENERGY", "key":  1},
    { "name":  "IS_SITTING", "key":  2},
    { "name":  "THIRST", "key":  3},
    { "name":  "IS_VISITING_STALL", "key":  4}
  ],
  "interruption_handlers": [
    { "name":  "PLAYER_SHOUTS", "key":  0},
    { "name":  "MARKET_OPENS", "key":  1}
  ]
}
```

The file contains two arrays: `"entity_states"` and `"interruption_handlers"`. The `"entity_states"` array defines the state properties available to entities in a game. Each entry has a `"name"` field (a human-readable string) and a `"key"` field (an integer). The `"name"` is the string that other configuration files use when referencing this state property, such as when specifying preconditions or effects in action definitions. The `"key"` attribute is the integer that the framework uses internally. Each `"key"` must be unique within the `"entity_states"` array.

The `"interruption_handlers"` array defines the interruption events that the framework can process. Each entry has the same `"name"` and `"key"` structure. The `"name"` is used in entity configuration files to map interruption events to response sequences. Each `"key"` must be unique within the `"interruption_handlers"` array.

The names defined in this file are strings chosen by the developer. The framework assigns no built-in meaning to any name; "CAPACITY" and "ENERGY" are conventions chosen for the marketplace demo. All other configuration files reference these names, which are resolved to their corresponding integer keys at load time. Because of this, the state schema must be loaded before any other configuration file.

A.3 Environmental Conditions Format

The environmental conditions file defines the game conditions that the framework can query from a game engine during runtime. These conditions represent aspects of the game world that exist outside of any individual entity's state, such as weather, time of day, or whether a particular area of the game world is accessible.

The `"environmental_conditions"` array defines each condition tracked by the framework. Each entry of this array has three fields. The `"condition_key"` field is a unique integer identifier for the condition. It must be unique across all environmental conditions in a game. The `"name"` field is a human-readable string used to reference this condition in other configuration files. When a state operation in an action or transition precondition uses `"EN-`

Listing A.2: Example of an environmental conditions configuration file.

```
{
  "environmental_conditions": [
    {
      "condition_key": 0,
      "name": "IS_MARKET_OPEN",
      "update_frequency_ms": 2000
    },
    {
      "condition_key": 1,
      "name": "WEATHER",
      "update_frequency_ms": 5000
    }
  ]
}
```

VIRONMENT" as its "target_id_name" and specifies a "state_key_name", that name must match the "name" of an environmental condition defined here.

The "update_frequency_ms" field specifies the minimum time in milliseconds between queries to game engine systems for this condition's current value. The framework caches the value between queries and only re-queries game engine systems when the cached value becomes stale (that is, when the elapsed time since the last query exceeds this frequency). Higher values reduce the number of queries but mean the framework may operate on slightly outdated information. The appropriate value depends on how quickly the condition changes in a particular game. In the listing above, market status is checked every 2 seconds while weather is checked every 5 seconds, reflecting that weather changes less frequently.

A.4 Action Definitions Format

The action definition file specifies all actions that NPCs can perform. Each action specifies preconditions that must be satisfied before execution, state changes that occur at different points during execution, and timing and interruption parameters. This is the most complex configuration file in the framework.

Listing A.3 defines a "walk to sit" action that an NPC performs to walk toward an entity (such as a bench) and sit on it. The action requires a target entity to have available capacity and to be within a specified distance of the NPC. When the action begins, an immediate effect reserves a seat by decrementing the target entity's capacity. When the action completes, completion effects decrement the NPC's energy and thirst values. These values are designer-configured to reflect the intended cost of performing the action. If the action is interrupted before completion, an interruption effect restores the target entity's capacity. The action is marked as resumable, meaning the framework may attempt to continue it after an interruption.

Listing A.3: Example of an action configuration file.

```
{
  "timeout_check_interval_ms": 500,
  "actions": [
    {
      "action_id": 115,
      "action_name": "walk_to_sit",
      "action_duration_ms": 10000,
      "action_timeout_ms": 60000,
      "interruption_behavior_name": "RESUMABLE",
      "preconditions": [
        {
          "target_id_name": "ENTITY",
          "state_key_name": "CAPACITY",
          "operation_name": "GREATER_THAN",
          "value": 0
        },
        {
          "target_id_name": "DISTANCE_TO_ENTITY",
          "state_key_name": "",
          "operation_name": "LESS_THAN",
          "value": 41
        }
      ],
      "immediate_effects": [
        {
          "target_id_name": "ENTITY",
          "state_key_name": "CAPACITY",
          "operation_name": "DECREMENT",
          "value": 1
        }
      ],
      "completion_effects": [
        {
          "target_id_name": "SELF",
          "state_key_name": "ENERGY",
          "operation_name": "DECREMENT",
          "value": 1
        },
        {
          "target_id_name": "SELF",
          "state_key_name": "THIRST",
          "operation_name": "DECREMENT",
          "value": 1
        }
      ],
```

```
      "interruption_effects": [
        {
          "target_id_name": "ENTITY",
          "state_key_name": "CAPACITY",
          "operation_name": "INCREMENT",
          "value": 1
        }
      ]
    }
  ]
}
```

Global Settings

The `"timeout_check_interval_ms"` field controls how frequently the framework checks whether any active action has exceeded its maximum timeout. This value applies to all actions defined in the file. Shorter intervals detect expired timeouts more precisely but result in more frequent checks. A value of 500 milliseconds provides a reasonable balance for most use cases.

Action Fields

Each entry in the `"actions"` array defines a single action with the following fields. The `"action_id"` field is a unique integer identifier for the action. It must be unique across all actions in a game and must be non-negative. Other configuration files reference actions by this identifier. The `"action_name"` field is a human-readable string used for logging and debugging. It has no effect on framework behavior.

The `"action_duration_ms"` field specifies the expected execution time of an action in milliseconds. This value is communicated to game engine systems when the action begins, allowing the engine to manage the action's rendering. The value must be greater than zero.

The `"action_timeout_ms"` field defines the maximum time the framework will wait for game engine systems to signal that an action has completed. If this time is exceeded, the framework completes the action automatically to prevent the NPC from remaining indefinitely in a waiting state. This value must be greater than `"action_duration_ms"`.

The `"interruption_behavior_name"` field specifies how the action behaves when interrupted by a game event. It accepts two values: "RESUMABLE" indicates that the framework may attempt to continue the action from where it was interrupted, while "NON_RESUMABLE" indicates that the action must restart from the beginning if attempted again. The full interruption and resumption mechanisms are described in Section 3.3.4 and Section 3.3.5.

State Operations in Actions

Preconditions and effects are defined using state operations (Section 3.1.2). Each state operation specifies four fields: `"target_id_name"`, `"state_key_name"`, `"operation_name"`, and `"value"`.

The "target_id_name" field specifies which type of target the operation applies to. This field accepts the following four values: "SELF" targets the NPC's own state. "ENTITY" targets the state of the entity that the NPC is interacting with. "ENVIRONMENT" targets a game environmental condition defined in the environmental conditions file. "DISTANCE_TO_ENTITY" evaluates the spatial distance between the NPC and a target entity. If any precondition in an action targets "ENTITY" or "DISTANCE_TO_ENTITY", the framework treats the action as entity-requiring, meaning that the entity identification process (Section 3.3.3) must locate a suitable target entity before execution can begin.

The "state_key_name" field identifies which property to access on the target. For "SELF" and "ENTITY" targets, this must be a name defined in the "entity_states" array of the state schema file. For "ENVIRONMENT" targets, this must be the "name" of a condition defined in the environmental conditions file. For "DISTANCE_TO_ENTITY" targets, this field must be left as an empty string because distance is computed from entity positions queried from the game engine rather than read from stored state.

The "operation_name" field defines what operation to perform. Preconditions use comparison operations: "EQUALS", "NOT_EQUALS", "GREATER_THAN", and "LESS_THAN". These evaluate the state property against the specified value and return true or false. Effects use modification operations: "SET", "INCREMENT", and "DECREMENT". These change the state property by the specified value. Using a comparison operation where a modification operation is expected (or vice versa) is a configuration error.

The "value" field provides the integer operand for the operation. For a comparison operation, it is the value that the state property is compared against. For a modification operation, it is the amount by which the state property is changed.

Effect Types

Actions define three types of effects, each applied at a different point during execution. The "immediate_effects" array contains state changes applied when an action begins, before a game engine starts the action's visual presentation. In the Listing A.3, the immediate effect decrements the target entity's capacity by one, reserving a seat on the target entity when an NPC begins walking toward it.

The "completion_effects" array contains state changes applied after the action completes. In the Listing A.3, the completion effects reduce an NPC's energy and thirst, reflecting the cost of performing the sitting action.

The "interruption_effects" array contains state changes applied if the action is interrupted before it completes. These effects typically reverse the changes made by immediate effects to maintain state consistency. In the Listing A.3, the interruption effect increments the target entity's capacity by one, releasing the reserved seat. Without this reversal, the target entity would permanently lose a seat every time an NPC is interrupted while attempting to sit.

Ensuring consistency between immediate effects and interruption effects is a developer responsibility. If an immediate effect decrements some value, the corresponding interruption effect should increment it by the same amount. The framework does not enforce this relationship automatically.

Distance Values

The `"value"` in a "DISTANCE_TO_ENTITY" precondition is expressed in the coordinate units of a game engine. In the Listing A.3, the value of 41 reflects the Manhattan distance units used in the Unity marketplace demo. Developers working with Unreal Engine must account for its use of centimeters as the base unit, which typically requires larger distance values for equivalent spatial ranges. Incorrect distance values can cause NPCs to fail precondition checks when visually they appear to be close enough, or to pass checks when visibly they are too far away.

A.5 Behavior Sequences Definition Format

The behavior sequences definition file specifies behavioral patterns as directed graphs formed by nodes connected by transitions. Each behavior sequence defines the structure of an NPC's behavior: Which actions to perform, in what order, and under what conditions.

Listing A.4 defines a "visit stall" behavior sequence representing the behavioral pattern an NPC follows when visiting a marketplace stall. The behavior sequence begins at node 1 (an action node), then proceeds to node 2 (a nested sequence node that executes a separate behavior sequence). After the nested sequence completes, the behavior sequence evaluates the outgoing transitions from node 2. The transition to node 3 has a precondition requiring sunny weather, while the transition to node 4 has no preconditions and is therefore always available. When the weather is sunny, both transitions are valid and the memory-driven selection algorithm chooses between them. When the weather is not sunny, only the transition to node 4 is available. Both paths lead to node 5, which is the end node that marks completion of the sequence. Figure A.1 illustrates this structure as a directed graph.

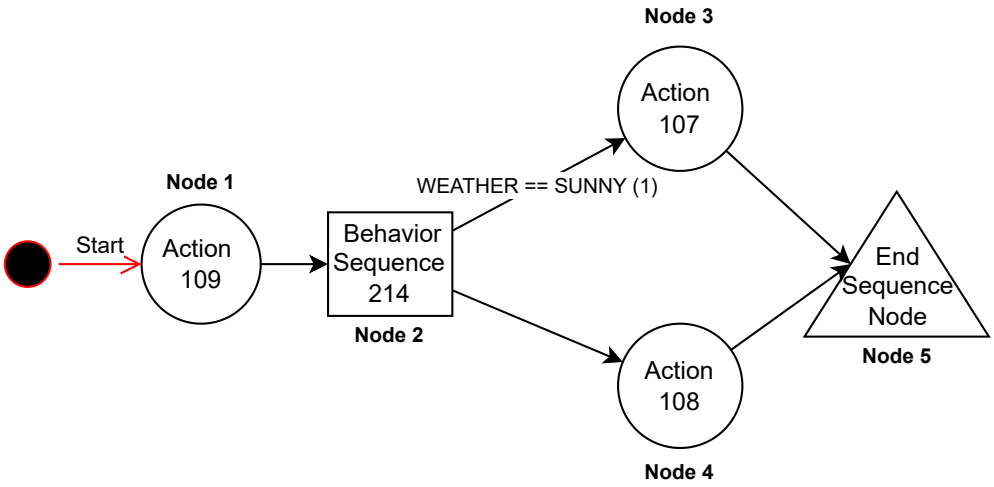

Figure A.1: Example of a behavior sequence.

Listing A.4: Example of a behavior sequence configuration file.

```
{
  "sequences": [
    {
      "sequence_id": 213,
      "sequence_name": "visit_stall",
      "entry_point_node_id": 1,
      "nodes": [
        {
          "node_id": 1,
          "node_type": "ACTION",
          "target_action_id": 109
        },
        {
          "node_id": 2,
          "node_type": "SEQUENCE",
          "target_sequence_id": 214
        },
        {
          "node_id": 3,
          "node_type": "ACTION",
          "target_action_id": 107
        },
        {
          "node_id": 4,
          "node_type": "ACTION",
          "target_action_id": 108
        },
        {
          "node_id": 5,
          "node_type": "END"
        }
      ],
      "transitions": [
        {
          "transition_id": 1,
          "from_node_id": 1,
          "to_node_id": 2,
          "preconditions": []
        },
        {
          "transition_id": 2,
          "from_node_id": 2,
          "to_node_id": 3,
```

```
            "preconditions": [
              {
                "target_id_name": "ENVIRONMENT",
                "state_key_name": "WEATHER",
                "operation_name": "EQUALS",
                "value": 1
              }
            ]
          },
          {
            "transition_id": 3,
            "from_node_id": 3,
            "to_node_id": 5,
            "preconditions": []
          },
          {
            "transition_id": 4,
            "from_node_id": 2,
            "to_node_id": 4,
            "preconditions": []
          },
          {
            "transition_id": 5,
            "from_node_id": 4,
            "to_node_id": 5,
            "preconditions": []
          }
        ]
      }
  ]
}
```

Sequence Fields

Each entry in the `"sequences"` array defines a single behavior sequence. The `"sequence_id"` field is a unique integer identifier for the behavior sequence. It must be unique across all behavior sequences in a game and must be zero or greater. Other configuration files and nested sequence nodes reference behavior sequences by this identifier. The `"sequence_name"` field is a human-readable string used for logging and debugging. It has no effect on framework behavior. The `"entry_point_node_id"` field specifies which node execution begins at when the behavior sequence starts. It must match the `"node_id"` of one of the nodes defined in the `"nodes"` array.

Nodes

The `"nodes"` array defines the steps in a behavior sequence. Each node has a `"node_id"` (a unique integer within this sequence) and a `"node_type"` that determines its role. An "ACTION" node represents a step where the NPC performs an action. It includes a `"target_action_id"` field that must reference an action defined in the actions configuration file. When the framework reaches this node, it initiates the action execution process described in Section 3.3.3. A "SEQUENCE" node represents a step where execution transfers to another behavior sequence. It includes a `"target_sequence_id"` field that must reference a behavior sequence defined in the behavior sequences configuration file. The referenced behavior sequence is pushed onto the NPC's sequence stack and executes to completion before the current behavior sequence continues. An "END" node marks the completion of the behavior sequence. Unlike ACTION and SEQUENCE nodes, it requires only the `"node_id"` and `"node_type"` fields. When the framework reaches an end node, the behavior sequence is removed from the NPC's sequence stack.

Transitions

The `"transitions"` array defines the directed edges between nodes. Each transition has a "transition_id" (unique within this behavior sequence), a `"from_node_id"` specifying the source node, a `"to_node_id"` specifying the destination node, and a `"preconditions"` array. The preconditions use the same state operation format described in Section A.4. All preconditions in the array must evaluate to true for the transition to be available. An empty preconditions array means the transition is always available when the source node completes.

In Listing A.4, node 2 (a nested sequence node) has two outgoing transitions: one to node 3 with a precondition requiring the environmental condition "WEATHER" to be equal to 1, and one to node 4 with no preconditions. This branching structure is shown in Figure A.1. Because the framework represents all state values as integers, environmental condition values are developer-defined; in this example, a value of 1 could represent sunny weather while other values represent different conditions. After the nested sequence at node 2 completes, the framework evaluates both transitions. If the weather condition equals 1, both transitions are valid and the memory-driven selection algorithm (Section 3.1.5) chooses between them. If the weather condition has any other value, only the transition to node 4 is valid. If no outgoing transitions from a node have their preconditions satisfied, the sequence enters the failed state and the framework activates fallback recovery (Section 3.3.6).

A single node can have multiple outgoing transitions to different destinations, each with its own preconditions. This structure allows the same sequence to produce different behavioral outcomes depending on current game conditions.

A.6 Entity Configuration Format

Entity configuration files define the entities that participate in a game. The framework supports two entity types: framework entities represent objects in the game world that NPCs can interact with (such as benches, stalls, or doors), while behavioral entities represent NPCs

with decision-making capabilities. Each entity is defined in its own configuration file and registered with the framework at runtime.

A.6.1 Framework Entities

Listing A.5 defines a bench in the marketplace scenario. The bench is a static entity (it does not move), starts with a capacity of 2 (allowing two NPCs to use it simultaneously), and accepts nine actions that NPCs can perform on it.

Listing A.5: Example of a framework entity configuration file.

```
{
  "entities": [
    {
      "entity_type": "FRAMEWORK",
      "entity": {
        "entity_id": 30,
        "entity_name": "bench_30",
        "is_static": true,
        "position_update_frequency_ms": 0,
        "accepted_actions_ids": [110, 111, 112, 113, 114, 115, 125,
           126, 127],
        "initial_state": {"CAPACITY": 2}
      }
    }
  ]
}
```

The `"entities"` array wraps entity definitions for consistency across configuration file types. Each file defines one entity. The `"entity_type"` field specifies whether this is a "FRAMEWORK" or "BEHAVIORAL" entity. The `"entity_id"` field is a unique integer identifier that must be zero or greater and unique across all entities (both framework and behavioral) in a game. The `"entity_name"` field is a human-readable string used for logging and debugging.

The `"is_static"` field indicates whether the entity moves during gameplay. Static entities (such as benches or buildings) do not need position updates. Non-static framework entities (such as a moving platform) require periodic position updates. The `"position_update_frequency_ms"` field specifies the minimum time between position queries to game engine systems for this entity. For static entities, this value is typically set to 0 because the entity's position does not change after registration. For non-static entities, this value controls how frequently the framework refreshes the entity's cached position.

The `"accepted_actions_ids"` field is an array of action identifiers that can be performed on this entity. Every identifier in this array must correspond to an action defined in the actions configuration file. This list determines which actions the framework considers when an NPC targets this entity. In Listing A.5, the bench accepts nine different actions, meaning NPCs performing any of those actions may select this bench as a target during the entity identification process (Section 3.3.3).

The "initial_state" field is an object mapping state property names to their starting values. Each name must be defined in the "entity_states" array of the state schema file. In Listing A.5, the bench starts with a capacity of 2, meaning two NPCs can use it simultaneously.

A.6.2 Behavioral Entities

Listing A.6 defines a visitor NPC in the marketplace scenario. The NPC follows behavior sequence 217 as its main behavioral pattern, has one interruption handler that responds to the "player shouts" event by executing behavior sequence 204, and has four fallback behavior sequences available for recovery. Its initial state includes tracking variables for sitting and stall-visiting status, as well as energy and thirst values that are modified by action effects during gameplay.

Listing A.6: Example of a behavioral entity configuration file.

```
{
  "entities": [
    {
      "entity_type": "BEHAVIORAL",
      "entity": {
        "entity_id": 12,
        "entity_name": "visitor_npc_12",
        "is_static": false,
        "position_update_frequency_ms": 2000,
        "accepted_actions_ids": [],
        "main_sequence_id": 217,
        "interruption_handlers": {"PLAYER_SHOUTS": 204},
        "fallback_sequences": [200, 201, 202, 203],
        "initial_state": {
          "IS_SITTING": 0,
          "IS_VISITING_STALL": 0,
          "ENERGY": 6,
          "THIRST": 6
        },
        "memory_limits": {
          "max_transition_memories": 10,
          "max_action_memories": 15,
          "max_interruption_memories": 3
        }
      }
    }
  ]
}
```

A behavioral entity shares the base fields described for framework entities ("entity_id" , "entity_name", "is_static", "position_update_frequency_ms", "accepted_actions_ids", "initial_state") and adds fields for decision-making.

Behavioral entities are typically non-static with a non-zero `"position_update_frequency_ms"` because NPCs move during gameplay. The `"accepted_actions_ids"` array is typically empty because NPCs perform actions on other entities rather than being action targets themselves. If an NPC should also be targetable by other NPCs (for example, an NPC that can be spoken to), its accepted action identifiers would be populated.

The `"main_sequence_id"` field specifies the primary behavioral pattern for an NPC. It must reference a behavior sequence defined in the behavior sequences configuration file. This behavior sequence is placed in the NPC's sequence stack during registration and is automatically restarted whenever the stack becomes empty (Section 3.1.4).

The `"interruption_handlers"` field is an object mapping interruption handler names to response behavior sequence identifiers. The keys must be names defined in the `"interruption_handlers"` array of the state schema file. The values must be behavior sequence identifiers defined in the behavior sequences configuration file. When a game event triggers an interruption, the framework looks up the corresponding response behavior sequence and pushes it onto the NPC's sequence stack, suspending the current behavior sequence (Section 3.3.4).

The `"fallback_sequences"` field is an array of behavior sequence identifiers used for recovery when the NPC's current sequence enters the failed state (Section 3.3.6). Each identifier must reference a behavior sequence defined in the behavior sequences configuration file. The framework randomly selects one fallback sequence from this array when recovery is needed. Fallback sequences should use actions with no preconditions to prevent cascading failures. Multiple fallback sequences provide variety even during recovery.

The `"memory_limits"` field controls how many memory records an NPC retains for each memory type. The `"max_transition_memories"` field sets the capacity for transition memories (which record past path choices through sequences), the `"max_action_memories"` field sets the capacity for action memories (which record past entity selections), and the `"max_interruption_memories"` field sets the capacity for interruption memories (which preserve context for action resumption). When a container reaches its limit, the oldest record is discarded to make room for a new one. Higher limits allow the selection algorithm to consider a longer decision history when choosing among valid options, resulting in more behavioral variety before an NPC revisits earlier choices. Lower limits cause the NPC to cycle through options more quickly. The appropriate values depend on the number of distinct options available at each decision point in the NPC's behavioral patterns.

B. Supplementary Material

The framework source code, the Unity demo project, and the Unreal Engine demo project are available for download at: `https://www.csd.uwo.ca/~ebuitron/` This page includes build prerequisites, setup instructions, and usage guidance for each download.