Список використаних джерел

1. Stuttard D. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws / D. Stuttard, M. Pinto. – Indianapolis : John Wiley & Sons, 2011. – 912 p.
2. Bunting S. EnCase Computer Forensics -- The Official EnCE: EnCase Certified Examiner Study Guide / S. Bunting. – Indianapolis : John Wiley & Sons, 2012. – 912 p.
3. Корченко О. Г. Захист інформації та кібербезпека в інформаційно-телекомунікаційних системах / О. Г. Корченко. – Київ : КПІ ім. Ігоря Сікорського, 2021. – 312 с.
4. National Vulnerability Database (NVD) [Електронний ресурс] / NIST. – Режим доступу: https://nvd.nist.gov/ (дата звернення: 22.05.2026).
5. OWASP. OWASP Top 10:2021. The Ten Most Critical Web Application Security Risks [Електронний ресурс] / OWASP // OWASP Foundation. – Режим доступу: https://owasp.org/Top10/ (дата звернення: 22.05.2026).

DOI 10.70286/EOSS-01.06.2026.004.217-231

SOFTWARE ARCHITECTURE AND IMPLEMENTATION OF A DATA-DRIVEN ADAPTIVE AI SYSTEM FOR 2D TACTICAL GAMES

Ziuziun Vadym
PhD, Associate Professor
ORCID: 0000-0001-6566-8798
Kovalchuk Oleksandr
Master's Deg. Stud.
ORCID: 0009-0001-5549-2116
Department of Technologies Management
Taras Shevchenko National University of Kyiv, Ukraine

Abstract. This article addresses the problem of limited replayability and the predictable behavior of non-player characters (NPCs) in modern 2D tactical simulation video games. The study examines the design and software implementation of a flexible, adaptive artificial intelligence (AI) system that operates without relying on asymmetric game balancing techniques («cheating»). The objective of the research is to develop the architecture and functional modules of a game capable of dynamically responding to a player's strategy in real time.

To construct the decision-making system, the Hierarchical Task Network (HTN) planning method was employed, integrated with utility functions for action evaluation and selection. Spatial and tactical analysis of the game environment was implemented using Influence Maps and the A* graph-based shortest pathfinding algorithm. The software implementation of the project was carried out using the modern game engine

Godot Engine 4.5, following a data-driven approach through hierarchical JSON configuration files. A modular architecture was developed, comprising the tactical analyzer, terrain manager, and config loader subsystems, which enables the computation of complex HTN plans within a background thread. This approach ensures high computational efficiency, supports realistic group behavior of enemy units, including flanking maneuvers, tactical retreats, and position holding, and enhances the overall depth and strategic complexity of the gameplay experience.

Key words: artificial intelligence, hierarchical task network (HTN) planning, tactical video games, Godot Engine, A* algorithm, influence maps, data-driven approach, decision-making, non-player characters (NPCs), game computation optimization.

Research implementation. The modern video game industry is undergoing a period of transformation in which the primary focus is shifting from purely visual aspects to the depth and quality of the player experience. In the genre of tactical games, the behavior of enemy non-player characters (NPCs) is a key factor influencing player engagement. As noted by Georgios N. Yannakakis and Julian Togelius, artificial intelligence (AI) in games has evolved beyond a mere pathfinding mechanism and has become a tool for generating dynamic content and creating adaptive game environments [1].

A significant challenge in many modern tactical games is the reliance on static decision trees or finite-state machines, which make enemy behavior predictable after only a few gameplay sessions [2]. Contemporary AI systems are expected not only to execute basic commands but also to plan complex sequences of actions and adapt to the player's strategy [3]. The development of a 2D tactical game featuring adaptive AI provides a means of addressing the issue of limited replayability while maintaining a high level of challenge and engagement for the player.

The objective of this study is to develop a 2D tactical video game project based on the implementation of an adaptive enemy AI system capable of making tactical decisions in response to dynamically changing gameplay conditions.

The software implementation of the 2D tactical game with adaptive artificial intelligence was developed using Godot Engine 4.5, which is based on a scene tree and node-oriented architecture. This model naturally aligns with a modular design approach and enables the system to be divided into independent components [1].

The project employs a data-driven architecture in which unit parameters, terrain data, campaign settings, and planning logic are stored in JSON files and loaded during initialization, enabling flexible configuration without requiring modifications to the source code [4; 5; 6]. Interactions between modules are implemented through Godot’s signal system, while the state-based behavior of individual components, particularly military units, is designed according to the Observer and State design patterns. This approach reduces tight coupling between system components and facilitates scalability and future expansion [7].

Configuration Layer and ConfigLoader

The central data-handling mechanism is ConfigLoader, a static utility module responsible for reading JSON files from the file system and converting them into internal Godot data structures (dictionaries and arrays).

It performs the following operations:

- validation of file existence and access permissions;
- parsing JSON using the engine's built-in parsing mechanism;
- returning fallback values in case of errors.

As a result, configurations (units.json, terrain.json, htn.json, campaign.json) are not hardcoded into the game logic but can be edited independently, which is a key advantage of the data-driven approach [4; 5; 6]. In addition, the use of tree-like structures (HTN) and dictionaries aligns with modern practices for storing AI data in a structured and formalized format [8].

Combat logic modules

1. BattleScene acts as a composition root node that instantiates the core components (terrain, battle manager, AI, camera, UI, and FX) and passes references between them.

2. BattleManager serves as the central battle controller. It is responsible for managing the phases DEPLOYMENT → BATTLE → RESOLUTION, tracking squads, triggering AI execution, and handling battle termination.

3. InputManager handles player interaction, including squad selection, movement and attack commands, as well as linear formation controls.

Artificial intelligence subsystem

The AI is implemented as a multi-layered structure that combines spatial analysis, fact generation, and action planning based on the Hierarchical task network (HTN) paradigm. This approach is widely used in modern game AI systems, particularly in strategic scenarios [9; 10; 11; 12; 13; 14]:

1. AIGeneral – the strategic core.

This is the main module responsible for executing the full planning cycle:

a) updating the influence map;
b) analyzing battlefield state;
c) performing HTN planning;
d) dispatching commands.

2. InfluenceMap – a spatial influence grid that stores positional metrics (density, strength, and threats) for both sides. It serves as the foundation for strategic analysis and flank evaluation.

3. TacticalAnalyzer – a sensory layer module that transforms raw data from the InfluenceMap into high-level abstract facts, such as force ratio, flank accessibility, and ranged combat advantage.

4.HTNWorldState – the factual knowledge base. This central dictionary represents the logical state of the system and is used by the HTN planner to evaluate preconditions and simulate effects.

5. HTNPlanner – a recursive planning module that decomposes the root goal into primitive tasks based on utility-based method evaluation. The resulting plan is converted into a linear sequence of actions [9; 15].

6. HTNDomain, HTNTactics, HTNJsonBuilder. These components represent the "knowledge base" of the AI, including task hierarchies, methods, tactical

definitions, and evaluation rules. The domain can be loaded from htn.json, making the AI partially configurable and easily editable [6; 8]. If the configuration is missing or invalid, the domain is constructed from code as a fallback.

7. CommandDispatcher – plan-to-action translation layer. After HTN planning, primitive actions are converted into concrete squad commands (movement, attack, flanking, holding positions). This forms a command queue later executed by the squad system.

8. EnemyAI – fallback logic. A simplified reactive AI used for testing or when HTN is disabled. Its behavior is limited to attacking the nearest enemy or moving toward the map center.

Spatial navigation module

Within TerrainManager, a custom A* grid (AStarGrid2D) is constructed with movement weights, where each cell has walkability and a traversal cost depending on terrain type. The path between two positions is computed as an optimal route on a graph, corresponding to the classical A* approach [16].

At the squad level, movement is handled by SquadPathfinding: it receives a path from TerrainManager, adjusts the start and end points to the nearest walkable cells, and returns a sequence of waypoints. For HTN-generated commands, the path is additionally sampled into a limited set of stable waypoints, reducing noise and improving squad movement predictability. This approach aligns with established practices in optimal pathfinding on weighted graphs [11; 16; 18].

Map subsystem

TerrainManager is responsible for procedural map generation, including:

- noise-based generation of elevation, forests, swamps, and rocky terrain;
- construction of rivers and roads;
- assignment of terrain walkability and movement costs;
- creation of an A* navigation grid.

The generated map serves as a unified data source for AI, navigation, and combat mechanics, making this subsystem one of the fundamental pillars of the overall game logic.

Interaction of software components

During gameplay, the interaction between software modules occurs in a continuous real-time cycle. Unlike turn-based systems, where AI logic is executed sequentially, the developed game operates with all subsystems running continuously and in parallel.

The cycle timing parameters are defined by three constants. First, the planner is activated 2,5 seconds after the start of the battle, allowing units sufficient time to reach their initial positions before the first tactical decision is made. Second, a full replanning cycle is executed every 5,0 seconds, which is enough for units to complete their current orders while still ensuring timely adaptation to changes in the tactical situation. Third, the Influence Map is rebuilt significantly more frequently, every 15 game frames, ensuring that spatial data remains up to date between two replanning cycles.

A critically important architectural decision is that HTN plan computation is executed in a background thread (WorkerThreadPool), fully avoiding any blocking of the main game thread. This ensures a stable frame rate even during recursive decomposition of complex tactical tasks. To initiate a new planning cycle, two conditions are checked simultaneously: the replanning interval has elapsed, and the previous planning process has already completed. If the background thread is still running, the current cycle is skipped, preventing the accumulation of an execution backlog.

Once the background thread completes execution, the resulting flat array of primitive tasks is passed to the CommandDispatcher, which converts it into queues of concrete commands for each unit. The tactical pause functionality halts all updates simultaneously, including Influence Map rebuilding, the replanning timer, and unit command execution, providing the player with full control over combat pacing [1; 16].

Design of algorithms and software interfaces

To ensure the functionality of the proposed architecture, core algorithms were designed and the structure of software interfaces was defined. Primary attention is given to the Hierarchical Task Network (HTN) decomposition algorithm and the interaction algorithm with the game's navigation system.

Description of algorithms for JSON configuration processing and AI planning

ConfigLoader is a core component of the data-driven architecture of the project and is responsible for reading external JSON configuration files and converting them into internal Godot structures (dictionaries and arrays). This enables flexible adjustment of game parameters without modifying the source code [5; 6]. The algorithm is implemented as two static functions: load_json(path, fallback) and load_json_dict(path, fallback), where path is the file location and fallback represents the default value used in case of an error.

Algorithm description (step-by-step):

1. File existence check. If the file at the specified path does not exist, the function returns the fallback value and logs a warning. This prevents crashes during game startup or when configuration files are missing in the build.

2. File opening and text reading. The file is opened in read mode. If it cannot be accessed (e.g., due to permission issues), the fallback value is returned. If successful, the entire content is read as a text string.

3. JSON parsing. The text is passed to a JSON parser. If a syntax error occurs, the function returns fallback and logs the line number and error message. This allows configuration issues to be localized at the loading stage [5; 6].

4. Returning data structures. If parsing succeeds, the result is returned as a generalized type (Dictionary / Array / primitives). For typical configuration cases, a helper function load_json_dict is used to ensure that the result is specifically a dictionary; otherwise, the fallback is returned. This maintains a unified access format for parameters and aligns with the use of dictionary-based and tree-like structures in AI and configuration systems [8].

5. Further use in subsystems. The resulting dictionaries are passed to corresponding modules (units, terrain, campaign, HTN domain), where parameters are accessed by keys, with default values used in case of missing entries. This approach ensures system stability even with incomplete configuration files.

Implementation of HTN trees in JSON

In the project, the HTN domain is stored in the htn.json file and loaded via HTNJsonBuilder.build_domain_from_json(). This enables a fully data-driven approach to the planner structure: tactical logic, preconditions, and evaluation rules are stored in data rather than hardcoded rules [4; 6; 9; 10; 15].

1. General JSON domain structure.

The file contains:

- version – format version;
- root – the name of the root compound task (for example, win_battle);
- compound_tasks – a dictionary of compound tasks, where each key is the name of a compound task and the value defines its methods.

2. Compound tasks.

Each compound task contains a list of methods. A method represents a single way of decomposing a compound task into subtasks.

Key method fields include:

- name – method identifier;
- precondition – a logical condition expressed as an evaluable expression (see section 4);
- precondition_desc and precondition_facts – textual description and a list of facts used for debug visualization;
- utility – a utility function definition (see section 5);
- subtasks – a list of subtasks, which may be either compound or primitive.

This implements the hierarchical structure of HTN: the tree is constructed from compound nodes that produce a plan through a sequence of subtasks [9; 15].

3. Primitive tasks.

A primitive task is a leaf node of the tree that corresponds to a concrete command for units.

Format:

```
{
  "primitive": {
    "name": "flank_weakest_side",
    "command": "flank_attack_weakest",
    "role": "cavalry",
    "expected_effects": {
      "idle_cavalry_count": 0
    }
  }
}
```

Supported fields include:

- command – the type of command executed by the CommandDispatcher;
- role – the squad role (infantry, ranged, cavalry, all);
- target – ціль у форматі fact, vec2 or an arbitrary parameter;
- expected_effects – the target specified as a fact, a HTNWorldState, applied during planning as a simulation of outcomes.

This enables the HTN algorithm to evaluate future states before actual execution.

4. Expression language in JSON (precondition / target / utility).

The JSON format uses a custom expression language with basic operations:

- logic: and, or, not
- comparison: ==, !=, <, <=, >, >=
- arithmetic: +, -, *, /
- functions: min, max, clamp, abs
- type casting: int, float, bool
- conditional operator: if
- fact access: fact
- constants: const

Example:

```
{"op": "if",
 "cond": {"op": "fact", "name": "battle_is_winning", "default": false},
 "then": 6,
 "else": 0}
```

These expressions are interpreted in HTNJsonBuilder._eval_expr() and read values from HTNWorldState, using default values if a fact is missing. This is particularly important for maintaining planning stability under incomplete data conditions [6].

5. Utility block and method selection.

The utility field can define:

- score – an expression that computes a numerical evaluation.
- reject_if – a formatted list used for debug output, helping interpret the score calculation.
- breakdown – a formatted list used for debug output, helping interpret the score calculation.
- desc і facts – metadata used for analysis and HTN visualization tools.

In practice, the JSON defines both the utility function and an explanation of tactical decisions, which is highly useful for testing, tuning, and balancing the system [10; 13].

6. Fallback methods.

If a method is marked with fallback: true, its preconditions are ignored and it is treated as «always valid».

This ensures that the tree contains a backup decision path (e.g., last_stand), preventing the planner from ending up without any available actions.

7. Conversion of JSON into internal objects.

HTNJsonBuilder:

- reads root and compound_tasks;
- recursively constructs HTNCompoundTask → HTNMethod → (HTNCompoundTask | HTNPrimitiveTask) structures;
- caches already constructed nodes to avoid duplication;
- converts JSON conditions into Callable functions that directly operate on HTNWorldState.

If the JSON is invalid or missing, a fallback implementation defined in code (HTNTactics) is used as a safety backup.

AI planner algorithm (HTN decomposition)

The central algorithm of the AI module is a recursive process that expands the root task win_battle into a flat sequence of atomic commands for units. Unlike the classical HTN approach, which selects the first valid method, the implemented planner uses a utility-driven selection mechanism, choosing the method that maximizes a weighted quality score among all methods satisfying the preconditions [10; 15].

The algorithm is implemented as a recursive function decompose(task, world_state, plan, depth) with the following parameters: the current task, a cloned world state world_state, a candidate plan an array of primitive actions accumulated so far), and the current recursion depth. The algorithm is described step by step:

1. **Depth limit check.** If depth > MAX_DEPTH (MAX_DEPTH = 50), the function immediately returns false. This prevents infinite recursion in cases of cyclic or excessively deep task hierarchies and guarantees that planning terminates within a bounded time.

2. **Processing a primitive task.** If the task is a primitive operator $o_i \in O$, the algorithm appends o_i to the plan array and applies the effects $Eff^+(o_i)$ і $Eff^-(o_i)$ to the current world_state, simulating the state transition after action execution [9]. The function returns true, indicating successful processing.

3. **Processing a composite task with utility selection.** If the task is a composite task t_c, алгоритм ініціалізує змінні $best\_score = -\infty$ та порожні best_plan і best_state. Далі для кожного методу $m_i \in M(t_c)$ виконується така послідовність:

- precondition check → if $Prem_i \not\subset$ world_state, the method is skipped;
- state cloning → independent copies are created (candidate_state ← world_state and candidate_plan ← plan). Cloning is critical, as it allows each method to be simulated in isolation without modifying the original state until a final selection is made;
- recursive subtask decomposition → for each subtask in $SubT(m_i)$, decompose(subtask, candidate_state, candidate_plan, depth + 1) is called recursively. If at least one subtask returns false, the method is considered invalid and is discarded;
- utility evaluation → if all subtasks are successfully decomposed, the following score is computed:

$$score(m_i) = utility(m_i, candidate_{state}) - penalty_{fallback}(m_i) - penalty_{length} \cdot |candidate_{plan}| \quad (4.1)$$

If score(m_i) ≤ 0, the method is rejected as tactically invalid. Otherwise, if score(m_i) > best_score, the current method becomes the new candidate: best_score ← score(m_i), best_plan ← candidate_plan, best_state ← candidate_state.

To ensure correct utility-based selection under parallel execution conditions, the decompose function operates exclusively on cloned state objects (candidate_state). Cloning is implemented via the clone() method of the HTNWorldState class, which performs a deep copy of the fact dictionary through a recursive traversal (equivalent to duplicate(true)). This prevents data races between iterations [1].

4. **Result finalization.** After iterating through all methods, if best_plan is not empty, the algorithm commits the selected result (world_state ← best_state, plan ← best_plan) and returns true. If no method passes selection, the function returns false, which triggers backtracking at a higher level or terminates planning with an empty plan.

5. **Completion and plan dispatch.** Once recursion fully expands the root task win_battle, the resulting plan contains a flat sequence of primitive tasks $\pi = \langle o_1, o_2, \ldots, o_k \rangle$. This array is passed to the CommandDispatcher, which translates each o_i into concrete commands for units of the corresponding role.

Because the HTN search does not stop at the first valid method but instead evaluates all possible methods and selects the optimal one, it enables tactically well-balanced decisions. For example, even if the Frontal Assault method is valid, the planner may choose Smart Flank Archers if it receives a higher utility score under the current battlefield conditions [10].

The data for this panel is provided by the planner: after the background computation is completed, AIGeneral copies last_evaluation_tree, last_planner_stats, and last_tactic_name into its internal fields.

Fig. 1 shows the HTN Tree Panel, a task decomposition tree for win_battle with utility scores for all methods and the selected tactic highlighted.

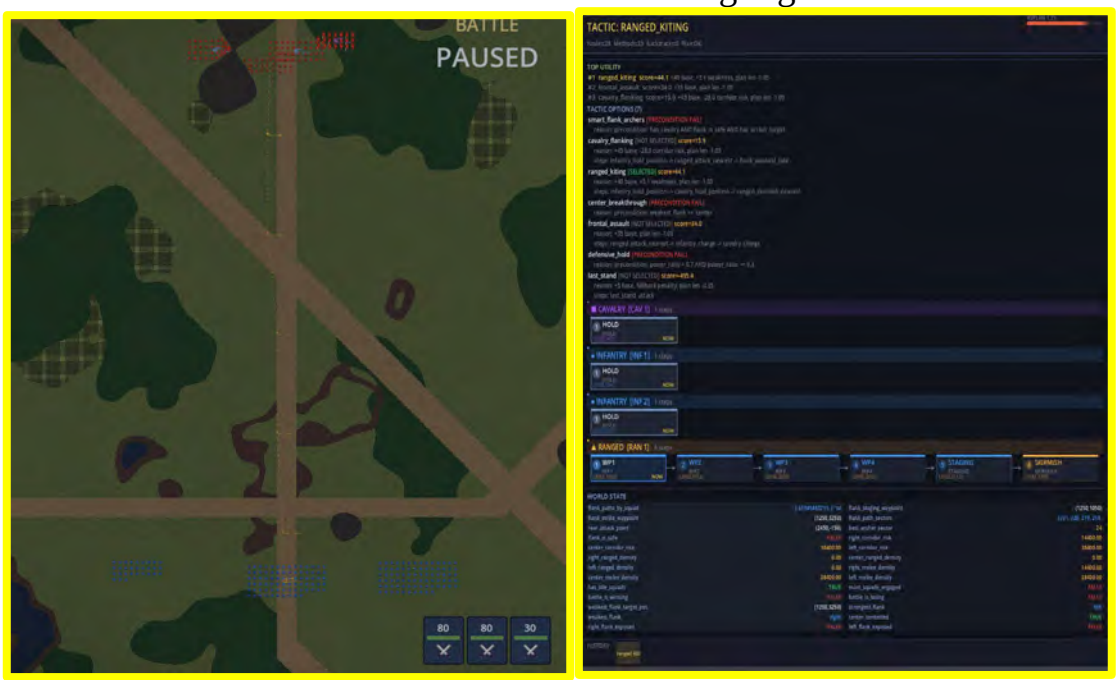

Figure 1. HTN Tree Panel – a task decomposition tree for win_battle showing utility scores for all methods and highlighting the selected tactic

The HTNTreePanel.gd script, within _process(), reads these fields and redraws the panel when Debug.show_htn_tree is enabled. Thus, the visualization directly reflects the result of decomposition without additional signals, although the plan_generated(plan) signal is also available for other debug overlays.

Spatial Navigation Algorithm (A* interaction)

After the HTN planner generates a movement command, the navigation subsystem uses the built-in AStarGrid2D of Godot, initialized in TerrainManager with consideration of cell walkability and terrain-based movement costs. Pathfinding is performed via TerrainManager.find_path, which returns a PackedVector2Array of points.

Movement steps:

1. The controller receives a target point in world coordinates and passes it to SquadPathfinding.request_path.
2. SquadPathfinding converts the coordinates into grid space, adjusts the start and end points to the nearest walkable cells, and calls TerrainManager.find_path (A*).
3. The resulting path is stored as current_path, while path_index indicates the active waypoint.
4. On each frame, _process invokes SquadMovement.process_moving, which moves the squad's anchor point along current_path via follow_path_step, taking into account unit speed and terrain movement modifiers.
5. Upon reaching the destination, _on_move_step_complete is executed, advancing the command queue or switching the squad to the IDLE state (without emitting a separate signal).

Implementation features:

Navigation is performed for the squad's anchor center. The formation is projected relative to the anchor, while individual units move toward their assigned formation_slot using a combination of spring-damper dynamics, separation forces, and obstacle avoidance mechanisms.

If an individual unit becomes stuck, a short bypass path is generated using TerrainManager.find_path. Afterward, the unit automatically returns to its designated formation position. Fig. 2 presents a corresponding visualization.

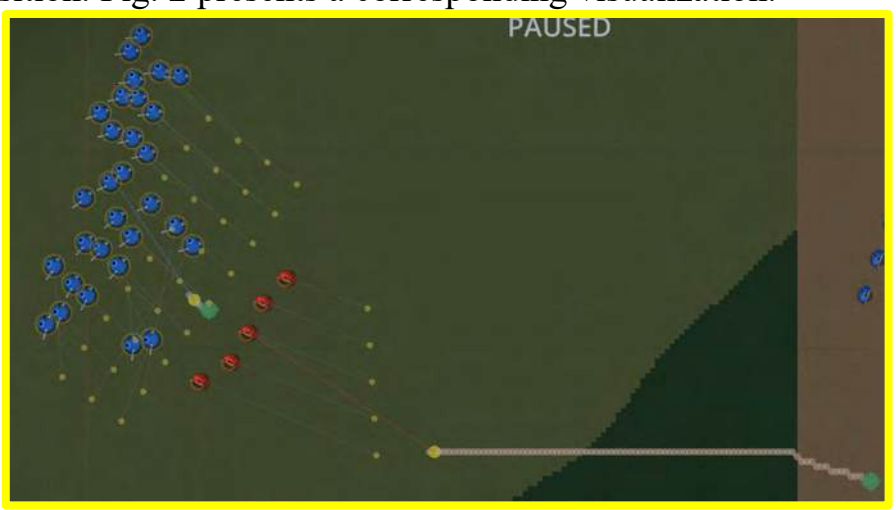

Figure 2. Visualization of A* paths in the game

Software interfaces (API)

To ensure modularity, interaction between the battle manager, input system, squads, and AI is implemented using Godot signals (observer pattern) [7]. Signals serve as an internal API for state synchronization without creating tight coupling between subsystems. The implementation includes the following event groups:

- combat cycle: phase_changed(new_phase), battle_ended(player_won), squad_deployed(squad) – manage phase transitions and notify the UI about changes in the battle state;
- player input: squad_selected(squad), squads_box_selected(squads), move_command(position), attack_command(target_squad), formation_mode_changed(active) – transmit player selections and commands without direct coupling to a specific UI implementation;
- squad state: morale_broken(squad), morale_rallied(squad), squad_destroyed(squad) – report critical morale changes and losses used to update interface indicators;
- AI system: plan_generated(plan), plan_failed – indicate HTN planning outcomes and are used in debug visualizations;
- debug subsystem: debug_state_changed – a unified trigger for updating overlays and interface panels.

Fig. 3 shows an example of squad signals in Godot Engine.

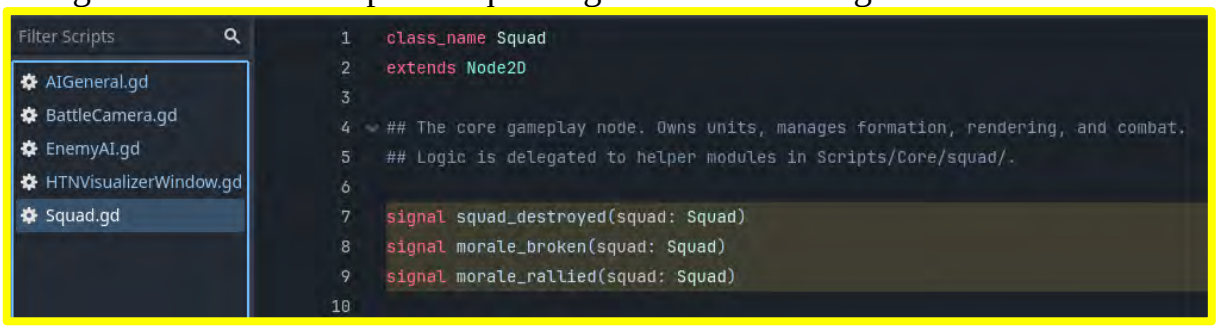

Figure 3. Example of Squad Signals in Godot Engine

User interface (UI)

The UI [19; 20] is decoupled from the game logic through CanvasLayer screen layers [1], ensuring independence from the camera and scene scale. During combat, multiple layers are used:

- in-combat HUD: top phase indicator, left deployment panel (unit buttons, squad size slider, scenario selection), formation mode indicator, battle results panel, and pause label;
- squad cards panel: located at the bottom center of the screen, displaying unit count, unit type, and a morale bar with color-coded thresholds; it updates every frame and reacts to morale_broken, morale_rallied, and squad_destroyed event;
- debug menu (F12): enables overlays and visualizations such as A* path visualization, HTNPlanner command tracing, enemy squad status display, and other diagnostic tools.

Fig. 4 illustrates the implemented HUD during the deployment phase, while Fig. 5–6 show the unit cards panel during combat and the debug overlay of enemy squad status, respectively.

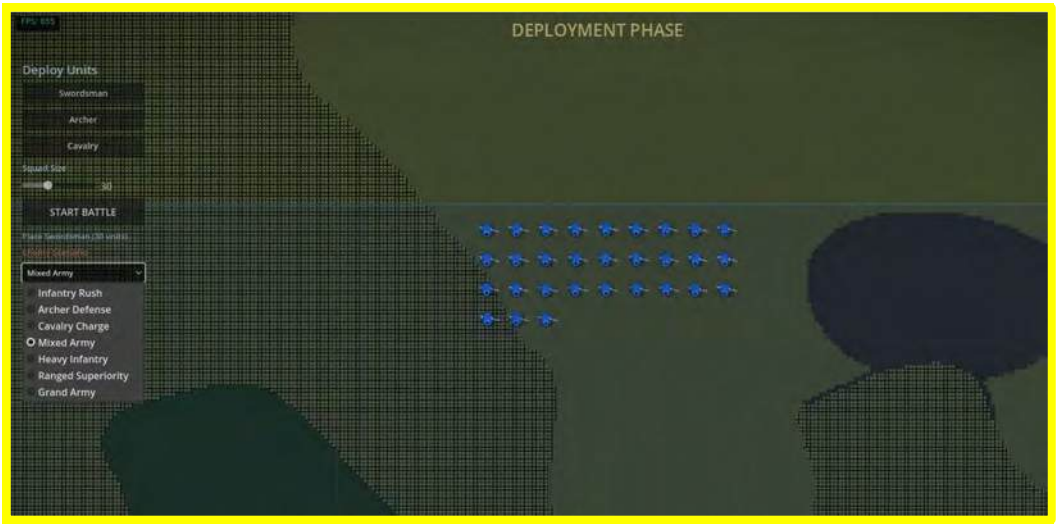

Figure 4. Implemented HUD during the deployment phase

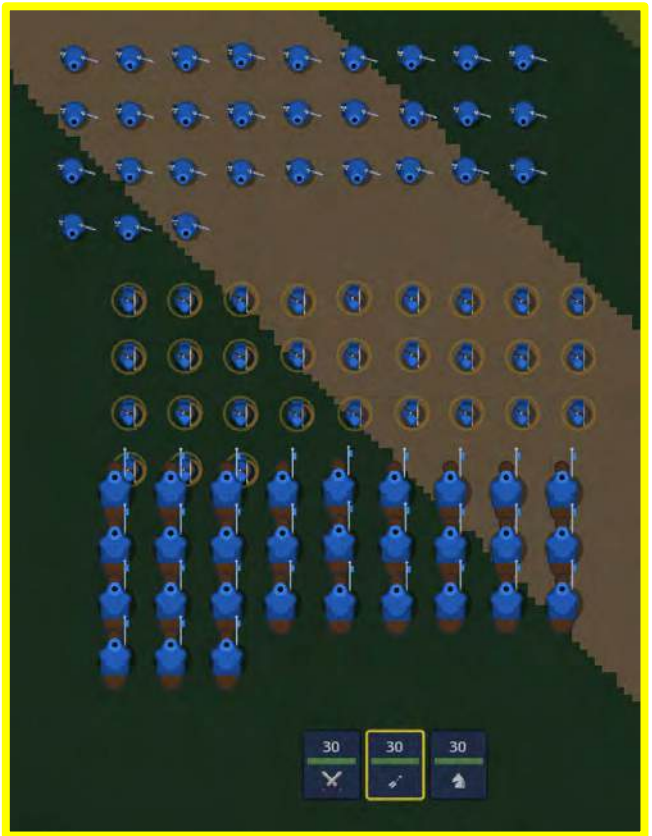

Figure 5. Unit cards panel interface during combat

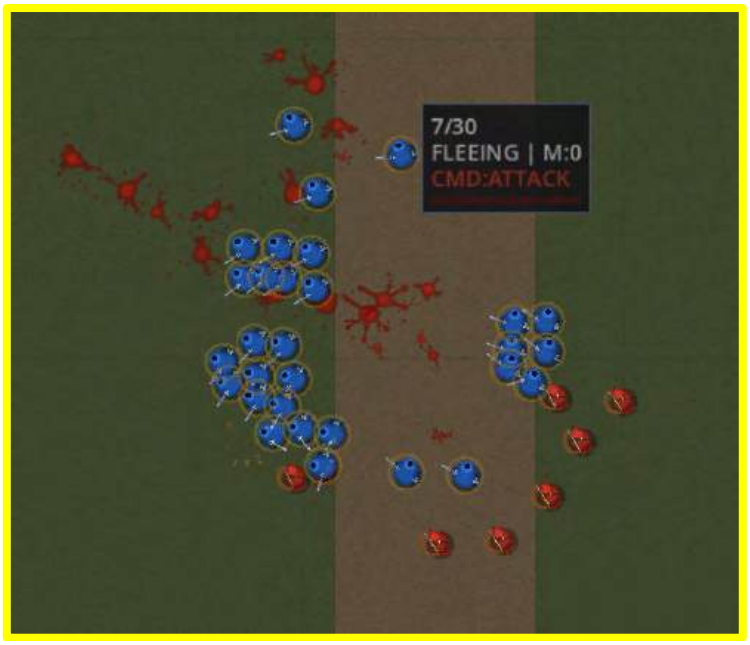

Figure 6. Debug overlay view of enemy unit status

Conclusion. As a result of the conducted research, a modular architecture for a 2D tactical video game with an adaptive artificial intelligence subsystem was successfully designed and implemented using the modern Godot Engine 4.5. The practical implementation confirmed the high efficiency of combining Hierarchical Task Network (HTN) planning methods with A* spatial navigation algorithms.

Through the development of the ConfigLoader component, a fully data-driven approach was achieved, allowing decision-making logic, unit parameters, and HTN domains to be separated from the game's source code and transferred into flexible external JSON configuration files. This engineering solution ensures high system scalability and enables real-time game balancing without the need for recompilation.

A critically important outcome of the work is the resolution of performance issues associated with the recursive decomposition of complex tactical tasks in real time. The integration of the planner with the engine's WorkerThreadPool system enabled background execution of optimal plan search processes, fully eliminating the risk of main-thread blocking and frame rate drops.

The use of Influence Maps, combined with the tactical analyzer, provided the AI with the ability to operate on abstract representations of battlefield states, generating tactically coherent and unpredictable group behaviors of enemy units, including flanking maneuvers, tactical retreats, and positional holding. The designed system demonstrates that a high level of player challenge and increased replayability can be achieved solely through the intellectual autonomy of AI algorithms, without relying on artificial numerical advantages or asymmetric balancing mechanisms.

References

1. (2024). Godot Engine Official Documentation: Introduction to the engine. Godot Foundation. URL: https://docs.godotengine.org/en/stable/getting_started/introduction/index.html

2. Buro, M. (2003). Real-Time Strategy Games: A New AI Research Challenge. URL: https://skatgame.net/mburo/ps/RTS-ijcai03.pdf
3. Millington, I. (2019). AI for Games, Third Edition. CRC Press, 1014 p.
4. Bilas, S.A. (2002). Data-Driven Game Object System. Game Developers Conference (GDC). URL: https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf
5. (2024). Godot Engine Official Documentation: Saving and reading data (JSON). Godot Foundation. URL: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html
6. (2017). The JSON Data Interchange Syntax. Standard ECMA-404. Ecma International. URL: https://ecma-international.org/publications-and-standards/standards/ecma-404/
7. Nystrom, R. (2014). Game Programming Patterns. Genever Benning, 354 p. URL: https://gameprogrammingpatterns.com/
8. Nosenko, T., Mashkina, I., &Jaskevych, V. (2022). Zastosuvannja algorytmiv ta struktur danyh u shtuchnomu intelekti. Zbirnyk prac' Kyi'vs'kogo universytetu imeni Borysa Grinchenka, 15 s. URL: https://elibrary.kubg.edu.ua/54962/1/T_Nosenko_I_Mashkina_V_Yaskevych_FITM.pdf
9. Erol, K., Hendler, J., & Nau, D.S. (1994). HTN planning: Complexity and expressivity. Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), URL: https://www.cs.umd.edu/~nau/papers/erol1994htn.pdf
10. Kelly, J. P., Botea, A., & Koenig, S. (2007). Planning with Hierarchical Task Networks in Video Games. URL: https://icaps07-satellite.icaps-conference.org/workshop8/Planning%20with%20Hierarchical%20Task%20Networks%20in%20Video%20Games.pdf
11. Oleshko, T.A., & Loban, O.V. (2019). Vybir algorytmu poshuku optymal'nogo shljahu peredavannja danyh u rozpodilenij systemi. Visnyk Nacional'nogo universytetu «L'vivs'ka politehnika». Serija: Komp'juterni systemy ta merezhi, 905. S. 42-48. URL: https://science.lpnu.ua/sites/default/files/journal-paper/2019/sep/18542/182073vis905ksmzvichniy-42-48.pdf
12. Yannakakis, G.N., &Togelius, J. (2018). Artificial Intelligence and Games. Springer. 345 p. URL: https://gameaibook.org/book.pdf
13. Ziuziun, V., & Petrenko, N. (2025). AI-Enhanced System Design for Agile Sprint Management and Velocity Prediction. 2025 IEEE 5th International Conference on Smart Information Systems and Technologies (SIST), P. 1-6. https://doi.org/10.1109/SIST61657.2025.11139278
14. Ziuziun, V., & Petrenko, N. (2025). AI Solutions for Optimizing SCRUM: Predicting Team Performance. Bulletin of National Technical University «KhPI». Series: System Analysis, Control and Information Technologies, (2) 14, 85-89. https://doi.org/10.20998/2079-0023.2025.02.11
15. Nau, D., &Au, T.C., & Ilghami, O., & Kuter, U. et al. (2003). SHOP2: An HTN planning system. Journal of Artificial Intelligence Research, 20. P. 379-404. URL: https://jair.org/index.php/jair/article/view/10362

16. Patel, A. (2023). Amit's Thoughts on Pathfinding. Red Blob Games. URL: https://theory.stanford.edu/~amitp/GameProgramming/
17. Godot Engine Official Documentation: Using Navigation Server. Godot Foundation, 2024. URL: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationserver.html
18. Shostal, K., & Ziuziun, V. (2026). Model of Technological Maturity of Artificial Intelligence Use in Banking Information Systems. Proceedings of the 3rd International scientific and practical conference «Information Systems and Technology: Results and Prospects» (IST 2026), Kyiv, Ukraine, P. 176-180. URL: https://ir.library.knu.ua/handle/15071834/13507
19. Ziuziun, V., & Demchuk, G. (2025). Information Architecture and UI/UX Design of a Mobile Application for Coworking Spaces. Proceedings of the 3rd International Scientific and Practical Conference «Innovations in Science: From Theoretical Foundations to Practical Impact», November 24-26, 2025, Antwerp, Belgium, P. 102-111. https://doi.org/10.70286/EOSS-24.11.2025.002
20. Ziuziun, V., & Koziuk, Yu. (2025). Conceptual Modeling of an Information System for Professional Development and Promotion of Educational Services in the Field of Digital Design. Scientific journal Transactions of Kremenchuk Mykhailo Ostrohradskyi National University, 2025, 4(153), P. 156-163. https://doi.org/10.32782/1995-0519.2025.4.19.

РОЗРОБКА ІНТЕРАКТИВНОГО ПІДРУЧНИКА З ПРОГРАМУВАННЯ НА PYTHON

Головач Йожеф
д.т.н., проф.
Кафедра математики та інформатики
Закарпатський угорський університет ім. Ференца Ракоці II
Україна

Розвиток інформаційних технологій (ІТ) надає нові можливості у галузі розробки навчальних посібників, насамперед з інформатики. Важливим напрямом використання ІТ є розробка на їх основі підручників нового покоління. Традиційні підручники часто обмежені можливістю демонстрації практичних завдань та відсутністю інтерактивності. Інтерактивні навчальні ресурси поєднують теорію та практику, забезпечуючи активну участь студентів та підтримку індивідуального темпу навчання.

Мова Python сьогодні є найбільш популярною мовою програмування, яку однаково ефективно застосовують у науці, інженерії та освіті [1-3]. Нами розроблено інтерактивний онлайн підручник по вивченню мови Python. Підручник реалізовано у виді веб-додатку, для створення якого було використано набір технологій, що включає HTML, CSS, JavaScript і PHP [4-7].