210014557

CS4303 P2 Report

Artificial Intelligence

10.04.25

Overview

"OVER S∞∞N" is the second iteration of a 2D platformer game that implements comprehensive AI systems that manage enemy behaviors, navigation, and decision-making. The player controls a samurai character who must defeat enemies with distinctive behavioral patterns and navigate through a vertically oriented level to collect a golden coin and win.

The game has multiple levels of AI implementation: fundamental steering behaviors that are used as the foundation for more complex movement patterns, compound behaviors that merge multiple steering forces, a robust finite state machine (FSM) for decision-making and changing the states of the enemies, and pathfinding based on the A* algorithm with a custom optimised heuristic for navigation. The core gameplay loop challenges players to adapt to different enemy types, each with unique AI-driven behaviors that create engaging and dynamic combat interactions.

Design

Core AI Mechanics

The game has an AI-driven enemy system that implements four different enemy types. Every enemy has unique behavioral characteristics that create varied challenges and combat interactions for the player. The enemy types are:

- Aggressive enemies that persistently chase the player
- Mixed-behaviour enemies that maintain balance between active pursuit and patrol
- Platform-bound enemies that patrol a dedicated area
- Evasive enemies that chase away from the player, only attacking if the enemy is in reach

Every enemy movement is built on a physics-integrated steering movement where the steering behaviours generate forces that are then applied to the enemies, while the force accumulation creates a smooth, realistic movement. Moreover, multiple behaviours can be combined and used together with different weights assigned to them, creating unique combinations. Lastly, the enemies that use the BoundedWanderer behaviour are environment-aware and do not fall off the platform, unless they are chasing the player.

All the decision-making for the enemies is done with an FSM. Every enemy utilizes the FSM with its six core states:

- IDLE: Default resting state with periodic transitions to patrol
- PATROL: Movement within a defined boundary using steering behaviors
- CHASE: Active pursuit of the player using pathfinding or direct steering
- ATTACK: Executing attack animations when in range with collision detection
- HIT: Reaction to taking damage with temporary invulnerability
- DEAD: Terminal state with death animation

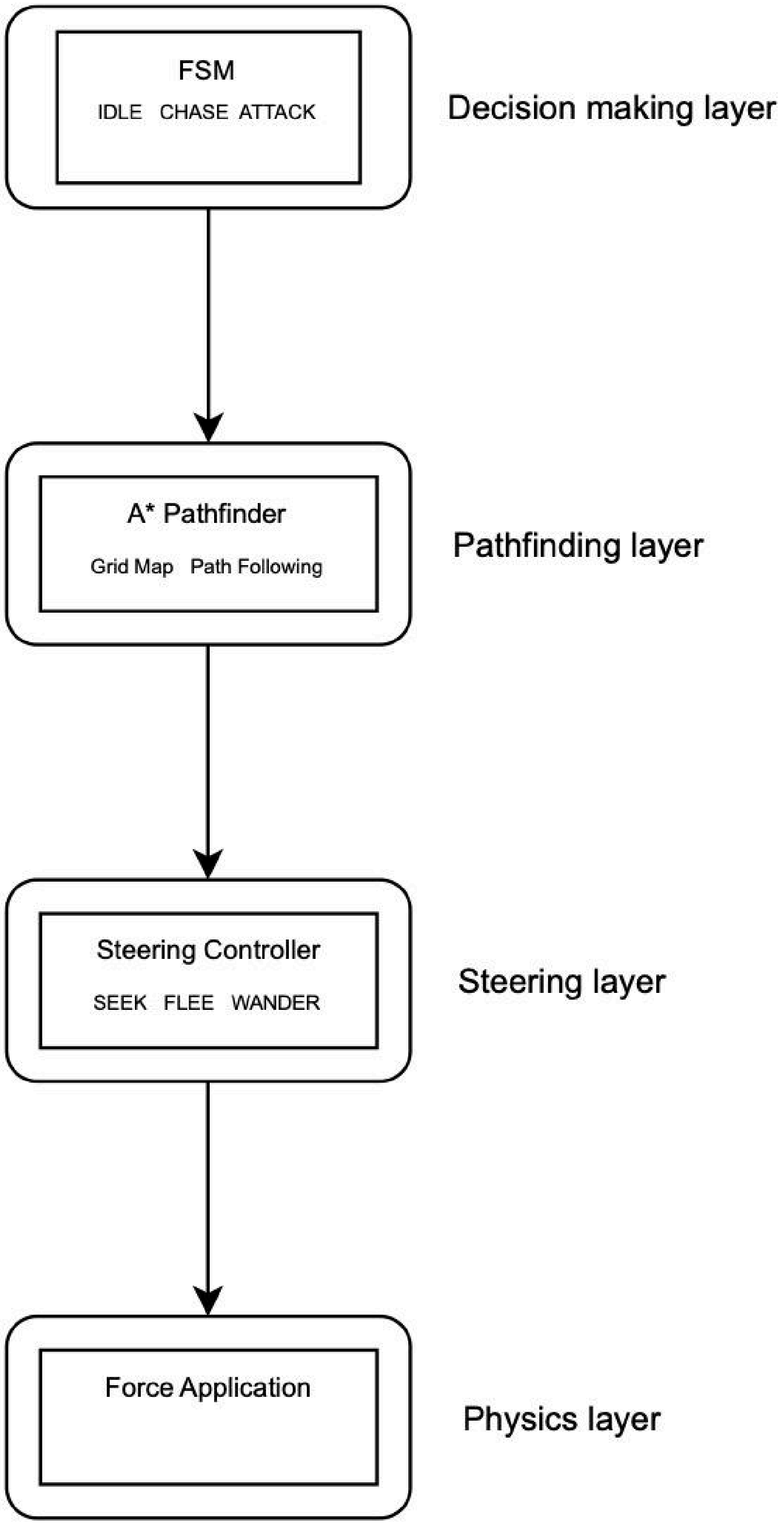
The FSM was built on top of the existing patrol behaviours that were in the game during the P1 development and has successfully allowed me to abstract the code and reuse it while creating unique enemies with their attributes. Additionally, implementing the enemy AI with an FSM rather than decision trees proved to be a more effective architectural choice due to the FSM excelling in real-time environments where state persistence is important, which allows enemies to maintain coherent behaviors over time rather than making isolated decisions each frame. This was particularly valuable for the platformer movement, where maintaining context (like "patrolling" or "attacking") creates more believable enemy behaviors and a stable game loop. Moreover, the FSM's explicit state transitions made debugging significantly easier, as I could clearly visualize which state each enemy occupied and why/when transitions occurred. The compact memory footprint of FSMs also offered performance advantages over decision trees, which would require evaluating numerous conditions each frame (especially with the later addition of Pathfinding). Lastly and most importantly, the FSM's modular design integrated seamlessly with the steering system, the existing blueprint for AI mechanics and pathfinding, which allowed me to leverage different navigation approaches based on the current context.

AI Architecture and Interaction

The AI system follows a structured architecture with a clear separation of responsibilities:

- **Steering Behaviors** are the foundation for all movement with interfaces for different behavior types. They generate forces that integrate with the existing physics system to create realistic movement patterns.
- The **Steering Controller** manages the application and weighting of multiple steering behaviors, which allows complex movement patterns to emerge from the multiplication of simpler behaviors.
- **FSM** handles decision-making through well-defined states and transitions, where each state encapsulates behavior patterns and transition logic based on game conditions.
- **Pathfinding** is responsible for the navigation capabilities through the A* algorithm modified for platformer mechanics. The algorithm converts between continuous world space and discrete grid space and finds the best path for the enemies to follow based on the custom heuristic.

The system is designed with extensibility in mind: interfaces and abstract classes allow me to add additional behaviors and states without significant restructuring. The core gameplay loop comes from the interactions between these AI systems and the existing physics engine. The interaction can be abstracted into four layers that utilize each other to form the structured approach that uses the FSM to determine the state of the enemy, then uses pathfinding to determine its navigation, and then uses the steering controller to apply the needed force (defined by the behaviour) using the physics engine. The detailed flow of interaction can be found in the diagram below.


FSM
IDLE CHASE ATTACK
Decision making layer
A* Pathfinder
Grid Map Path Following
Pathfinding layer
Steering Controller
SEEK FLEE WANDER
Steering layer
Force Application
Physics layer

Technical Description

Steering Behaviors Implementation

Implementing a modular steering system that efficiently interacts with the physics engine presented one of the primary challenges in the project. I designed a steering behavior interface that standardized how each behavior produced forces, enabling the creation of multiple behavior types that could be interchanged freely. For instance, the Seek behavior pursues targets through direct force calculation, determining the optimal direction to the target and applying maximum acceleration in that direction.

One of the more complex challenges was creating the Wander behavior, which adds natural, semi-random movement. This required generating forces that appeared random but maintained continuity between frames. The solution uses a continuously rotating displacement vector around a circle, with small random adjustments to the angle each frame. This method creates smooth, organic movement rather than generating completely random directions each frame, which would result in erratic, unrealistic behaviour. Based on the Wanderer behaviour, I have also implemented the BoundedWander, which required solving movement constraints unique to platformers. Unlike standard wandering behaviors, BoundedWander needed to generate random movement while preventing enemies from falling off platform edges. My solution monitors proximity to boundaries and applies corrective forces when enemies approach edges, creating invisible barriers. Finding the right balance between natural movement and edge avoidance took many iterations to get right and required careful force calibration. The result created enemies that patrol platforms intelligently without accidental falls, significantly enhancing gameplay with minimal computational overhead.

Compound Behaviors System

Creating a system to manage multiple simultaneous behaviors was essential for generating complex and organic enemy movement. I developed a steering controller that calculates weighted combinations of forces from different behaviors. The controller iterates through all active behaviors, calculates their individual force contributions, applies appropriate weights, and then combines them into a single resultant force.

The most challenging aspect was balancing these weights to create distinctive enemy personalities. For example, the evasive enemy combines Flee and Wander behaviors with specific weights, resulting in an enemy that maintains distance from the player while moving unpredictably. These weighted combinations proved much more effective than complex single behaviors, creating movements that would have been extremely difficult to script directly.

Decision-Making with FSM

Implementing a robust decision-making system required separating state logic from behavior implementation. The Finite State Machine (FSM) approach provided an organized framework that maintained clear state transitions. The system checks conditions and determines whether a state change is needed, handling clean state entry and exit to ensure behaviors are properly initialized and terminated.

Each state encapsulates its own logic, such as the ChaseStateHandler which manages pursuit behavior by monitoring the distance to the player and transitioning to attack when in range or returning to patrol when the player escapes. A significant challenge was integrating the FSM with the steering system. My solution was to have each state handler configure appropriate steering behaviors upon entry, clearing previous behaviors and setting up new ones specific to the current state.

This approach creates a clean separation between decision-making and movement mechanics while allowing them to work together seamlessly. The FSM structure also made debugging significantly easier, as I could identify the current state of each enemy with a visual indication.

My FSM follows this interaction flow:
States:
- IDLE
- PATROL
- CHASE
- ATTACK
- HIT
- DEAD

Transitions:
- IDLE → PATROL: Automatic timer-based transition
- IDLE → CHASE: Player detected
- PATROL → CHASE: Player detected
- CHASE → ATTACK: Enemy in range of player
- ATTACK → HIT: Enemy takes damage
- ATTACK → CHASE: Player out of range
- CHASE → HIT: Enemy takes damage
- HIT → CHASE: Hit animation complete, health > 0
- HIT → DEAD: Health <= 0
- CHASE → DEAD: Health <= 0

Initial State: IDLE/PATROL
Terminal State: DEAD

Pathfinding for Platformer Movement

Implementing pathfinding in a platformer environment posed unique challenges compared to traditional grid-based games. I developed a specialized A* implementation that accounts for platformer movement constraints. The system identifies valid movement options for each grid cell, considering horizontal movement and falling as possible actions while recognizing that vertical upward movement is impossible for the enemies.

The heuristic function was another critical element, tailored specifically for platformer movement. My asymmetric heuristic favors horizontal movement and falling down over moving upward (which enemies cannot do). Path smoothing further improved the natural appearance of movement by eliminating unnecessary waypoints while preserving critical path segments, particularly around height changes.

For my A* algorithm, I have devised a **modified Manhattan distance** heuristic:
$h(n) = |\Delta x| + \max(0, \Delta y)$

Δx = Horizontal distance to goal
Δy = Vertical distance to goal (only added if goal is below current position)

This heuristic never overestimates cost, as vertical distance is only included when reachable via falling, and if the player is above, Δy is ignored, preventing overestimation. Moreover, I belive that this heuristic is more efficient because it prioritizes horizontal alignment with the player first, automatically accounts for necessary downward drops, and for enemies on elevated platforms, the heuristic avoids impossible upward paths.

To justify my choice of the heuristic further, here is a comparison between the options I have considered when implementing my algorithm:

Heuristic type	***Suitability***
Manhattan	Overestimates for upward movement
Euclidean	Inefficient with platforms, accounts upward movement
Zero Heuristic	Degrades to Dijkstra (slow)
Modified Manhattan	Optimal

Another potential approach could be a Hybrid Euclidean heuristic $\sqrt{(\Delta x^2 + \Delta y^2)}$, but it would also overestimate if used for only horizontal movement.

My last challenge during development was integrating pathfinding with the FSM and steering systems. The solution was creating a PathFollow steering behavior that consumes path points, moving to the next waypoint when the current one is reached. The behavior primarily generates horizontal forces but allows vertical forces when falling is necessary, ensuring enemies navigate around obstacles and platform edges intelligently.

This integration creates intelligent enemies that can navigate the complex platformer environment while maintaining physically plausible movement patterns, significantly enhancing the game's challenge and realism. The pathfinding system periodically recalculates paths when the player moves significantly, ensuring that enemies can adapt to the player's changing position without excessive computational overhead. Lastly, to visually see the pathfinding process, the user can press **G** to see the grid and **P** to see the pathfinding algorithm work in real time.

Conclusion and Critical Review

My project successfully implements all four levels of the AI specification, creating enemies with distinct personalities and behaviors through layered AI systems. The integration of steering behaviors, compound behaviors, finite state machines, and pathfinding creates enemies that navigate the platformer environment intelligently while providing varied challenges for players.

As suggested in feedback from my previous submission, I've added multiple camera modes that players can switch between (by pressing 1 and 2), allowing for both a traditional fixed view and a character-following perspective. I've also enhanced the visual presentation with additional graphics that created a more visually appealing environment and composed a song to serve as the soundtrack.

The FSM architecture proved highly effective, offering a clear separation between decision-making and behavior execution. The most challenging aspect was developing the pathfinding system specifically for platformer environments, where vertical movement constraints create unique navigation challenges. I'm particularly satisfied with how the weighted compound behaviors create movement patterns that appear natural despite relatively simple rules. Additionally, the debugging visualization tools for pathfinding and steering behaviors were invaluable during development. Overall, the modular architecture provides a solid foundation for future enhancements while successfully delivering an engaging gameplay experience with intelligent, distinctive enemy behaviors.

With more time, I would implement dynamic difficulty adjustment where enemies learn from player behavior patterns and adapt their strategies accordingly. I would also expand the BoundedWander implementation to handle more complex platform configurations and develop group behaviors for coordinated enemy attacks.

Appendix

Borrowed Implementations (Graphics)

https://hugues-laborde.itch.io/pixelartsamurai?download
https://hugues-laborde.itch.io/pixel-art-character-pack

The UML diagram was made with draw.io