8-4-2025

Version: 1.0

OutcastSurvival

Detailed technical report for our game OutcastSurvival

Course
ICT.GP.AAI.V22_2425

Teacher
[redacted]

School & Education
Windesheim Zwolle HBO-ICT Software Engineering

Student
Bark, Ivan (s1169347)
[redacted]

hogeschool Windesheim

Version control

Version	Date	Description	Remarks
0.1	2-4-2025	First draft	Report structure
0.2	7-4-2025	Second draft	Theme and gameplay
1.0	8-4-2025	First version	Send for review

Distribution

Name	Role	Date	Version
[redacted]	Teacher	8-4-2025	1.0

hogeschool Windesheim

Contents

1. Introduction 3
   1.1. Theme 3
   1.2. Gameplay 3
2. Structure 4
   2.1. Overall Structure 4
   2.2. State machine 5
      2.2.1. State Structure 5
      2.2.2. Technical Structure 6
3. Techniques 8
   3.1. Behaviors 8
   3.2. Path Generation and Following 8
   3.3. Detection System 9
   3.4. Debugging and Visualization 9
   3.5. Sheep Flee Logic 9
4. Reflection 10
   4.1. Issues ..... **Error! Bookmark not defined.**
   4.2. Imporvements ..... **Error! Bookmark not defined.**
   4.3. Ideas ..... **Error! Bookmark not defined.**

References 11

1. Introduction

This report explains how we structured and built the game *OutcastSurvival* for this course. The goal of the assignment was to create a game that uses different AI techniques we learned, such as behaviors, pathfinding, fuzzy logic, and state machines. In this report, we describe the theme and gameplay, the structure of the game's systems, and the AI techniques used to make the game.

1.1. Theme

The theme of *OutcastSurvival* is about surviving in a world while being hunted. The player takes on the role of an outcast who has to survive by finding gold and food. Guards are trying to kill the player, and hunger is constantly increasing. This creates a mix of pressure, risk, and decision-making where the player has to keep moving, avoid being seen, and plan carefully.

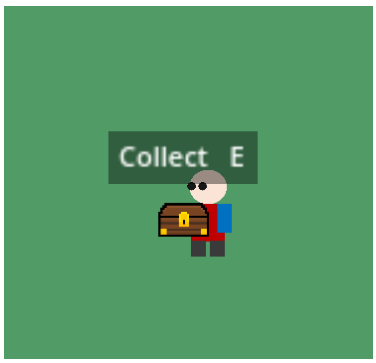
This survival theme gave a good reason to use different AI systems, like enemies with search and chase behavior, a fuzzy detection system that reacts differently depending on what the player is doing, and a hunger system that adds time pressure to the game.

1.2. Gameplay

OutcastSurvival is a top-down 2D survival game. The player wins by collecting 100 gold from gold chests placed around the map. But while doing this, the player also needs to watch out for guards that patrol and will attack if they spot the player. On top of that, the player gets hungrier over time and must hunt sheep to stay alive. If the player gets killed by a guard or dies from hunger, the game is over.

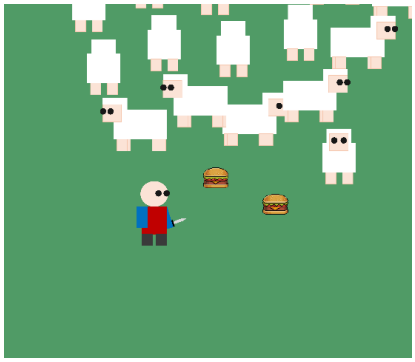
To win, the player needs to be careful, avoid guards, move strategically, and manage their hunger. Guards use a state machine to decide what to do, like searching, chasing, or attacking. Their detection is based on a fuzzy logic system that considers how far away the player is, the direction the guard is facing, and whether the player is sneaking or sprinting.

The rest of the report will explain how the game's systems are structured and what techniques were used to make the game mechanics and AI work.



Picture 1: Collecting gold

Picture 2: Attacked by guard



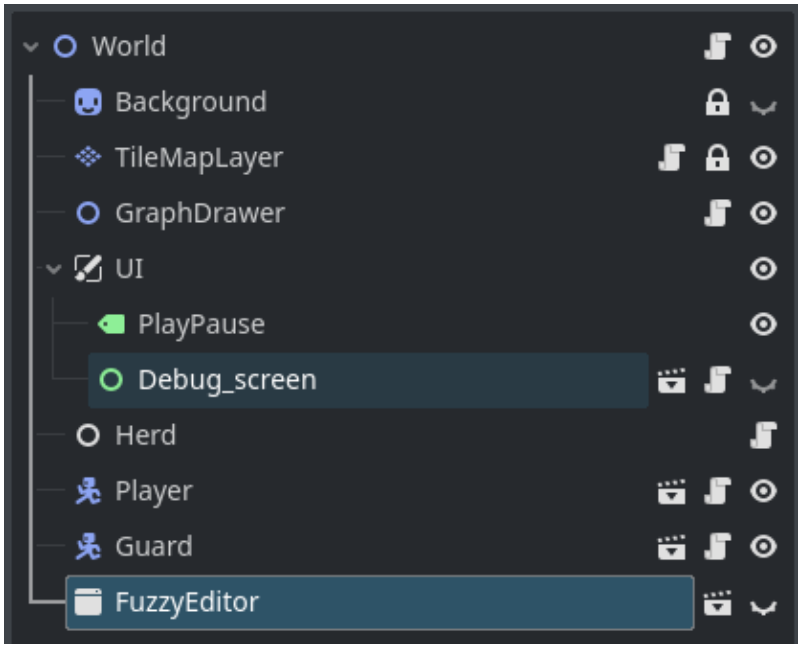
Picture 3: Hunting sheep

2. Structure

In this section, the structure of the game will be described. We'll explain how the main parts of the game are set up and how everything works together. First, the overall structure of the game will be described. Then, in more detail, there will be an explanation about the state machines used for AI, like the guards and other entities.

2.1. Overall Structure

We structured the files in folders roughly based on responsibilities. This helped us develop quickly as we only needed to look in the folder that matched the responsibilities. We also added the namespace based on the folder structure. As well as responsibilities, the folders roughly represent the nodes that are used in Godot. We also linked the classes and objects in the Godot editor; for example, the world scene looked like the picture below. The scroll next to nodes represents that there is a script attached. In our case, this is a C# class.



2.2. State machine

The guards in OutcastSurvival use a hierarchical finite state machine (HFSM) to manage their behavior. The state machine is split into root states, parent states, substates and its corresponding state machines, allowing guards to switch behavior depending on the player's actions and their own internal logic. A smaller version of this state machine is used for the player and its movement states like sneaking, walking and sprinting.

2.2.1. State Structure

The state structure of a guard looks like this:

```
Guard RootStateMachine
├── PatrolState
│   ├── IdleState
│   └── PatrolPathState
└── AlertState
    ├── PlayerLostState
    │   ├── SeekState
    │   └── SearchState
    └── PlayerDetectedState
        ├── ChaseState
        └── AttackState
```

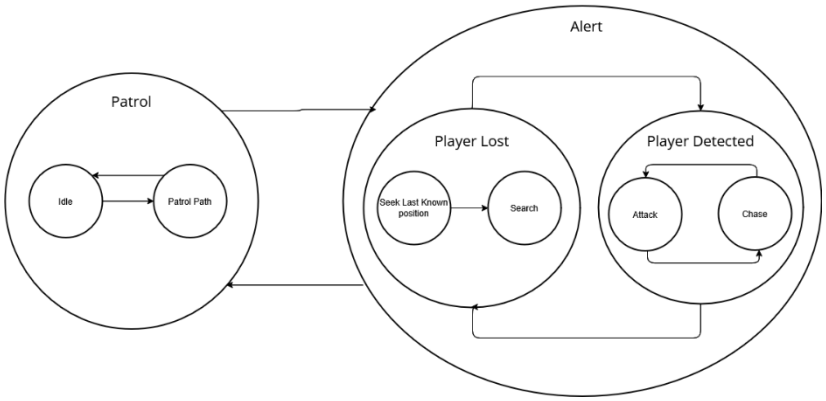


Diagram 1: State machine structure Diagram 2: State flow

This nested approach gives a better understanding of the flow of the states and it keeps the state machine from getting cluttered. This way each level can focus on its own logic. For example, Alert can focus on the high-level logic for Alert state and the transition between PlayerLost and PlayerDetected. It does not have to focus on having to attack, chase or search, only on its direct substates.

At the highest level, guards can be in either:

- PatrolState, where they roam the map without any awareness of the player.
- AlertState, which activates once the player has been detected.

Within PatrolState, guards alternate between:

- IdleState – standing still and waiting before moving again.
- PatrolPathState – following a predefined patrol route.

When transitioning to AlertState, the machine switches into one of two major alert branches:

- PlayerDetectedState – for when the player is currently seen.
    - ChaseState – moves toward the player.
    - AttackState – tries to kill the player if in range.
- PlayerLostState – for when the guard loses sight of the player.
    - SeekState – moves to the player's last known position.

- SearchState – performs a local search pattern.

A structure like this keeps things clear and organized.

2.2.2. Technical Structure

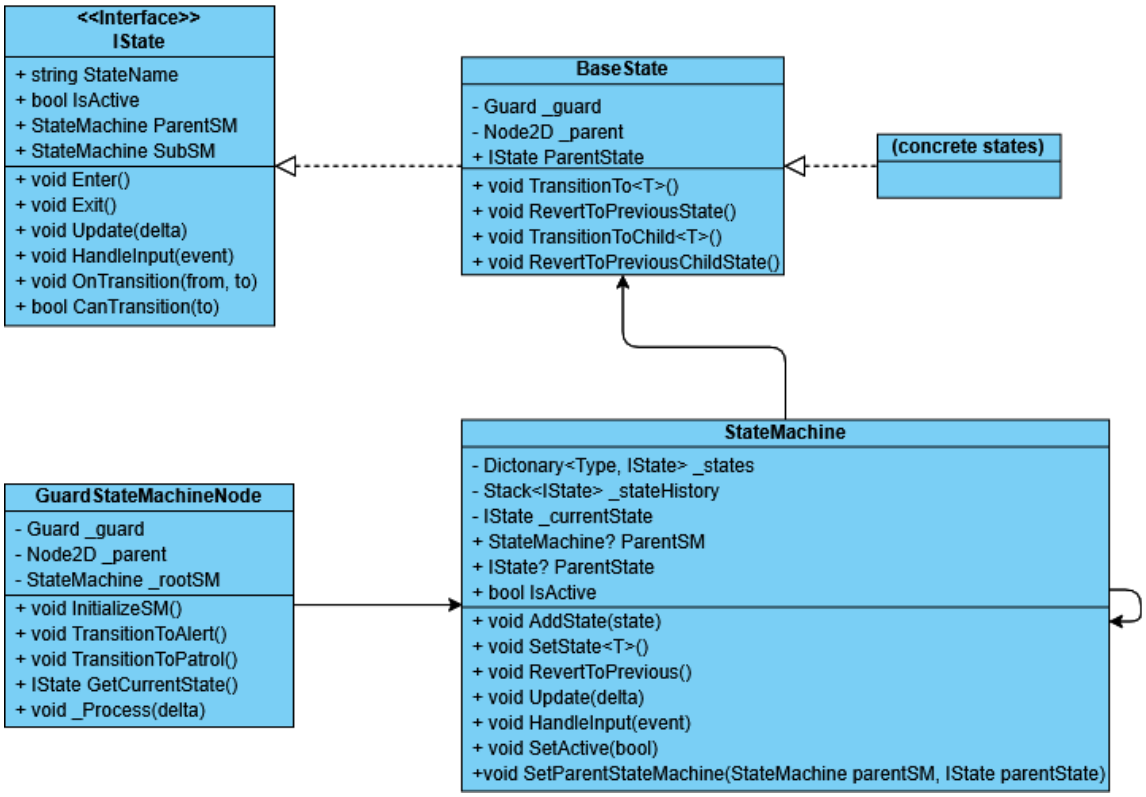


Diagram 3: Guard FSM Class Diagram

The FSM is implemented with a StateMachine class that handles transitions, history tracking, and updates. Each StateMachine can contain multiple IState objects and can also be nested inside a parent state. The GuardStateMachineNode is the node placed in the Godot Editor inside the Guard scene to connect the FSM to the guard.

Key parts of the system include:

- IState interface: Defines core methods like Enter(), Exit(), Update(), and HandleInput(). Each state implements this.
- BaseState class: A reusable abstract class that implements most of IState and provides helper methods like TransitionTo<T>().
- StateMachine class: Manages the currently active state, supports hierarchical nesting, and handles transitions and state history.
- GuardStateMachineNode: The node attached to each Guard in Godot. It initializes the full state machine structure, sets the starting state, and runs the update loop.

Each guard holds its own instance of a StateMachine, starting in PatrolState with IdleState as the substate. If the player is detected, the state machine transitions into AlertState, then into PlayerDetectedState, and finally into ChaseState or AttackState depending on range and

conditions. If the player escapes, it transitions into PlayerLostState, where it moves to the last seen position and searches the area before going back to patrol.

The player movement system uses the same structure as this.

3. Techniques

This section explains the main AI and algorithmic techniques used in OutcastSurvival. These techniques were implemented to create believable behavior for both enemies and passive entities, provide intelligent pathfinding, and handle player detection in a dynamic environment.

3.1. Behaviors

Different entities in the game exhibit distinct behaviors depending on their roles. The guards use a state machine to switch between actions like patrolling, chasing, or attacking the player. Each state controls a different part of their behavior:

- Patrol: stand still. (Did not have time to implement path patrol.
- Alert: Triggered when the guard detects the player.
- PlayerDetected: Guard has seen the player and starts to chase.
- PlayerLost: Guard lost track of the player and tries to find them again.
- Attack: Activated when the guard is close enough to deal damage.

Sheep use flocking behavior, which is built around three forces:

- Cohesion: Keeps sheep close together.
- Separation: Prevents sheep from overlapping or bunching up.
- Alignment: Makes sheep move in the same general direction as the group.

These forces help simulate natural group movement and make the sheep feel more alive. Each force has its own weight and radius, and they are combined into a final steering direction.

Furthermore, we implemented the following steering behaviors:

- Seek: seeks after the current position of the target, in this case it is used for the Chase behavior of the guards.
- Flee: seeks the opposite position of the current target, in this case it is used to flee from the player.
- Arrive: is used to go to the current target and stop there, in this case it is used for the search and seek last know position behavior of the guards.

3.2. Path Generation and Following

The game uses an A* pathfinding algorithm to generate paths across a tile-based map. This system allows sheep to move intelligently without walking through obstacles.

Key features:

- Graph-based map: Each walkable tile is a node in a graph.
- Waypoint-based movement: Entities follow the path one point at a time.
- Dynamic updates: If a target moves or a path becomes blocked, the path is recalculated.

- Obstacle avoidance: Entities detect nearby obstacles using a box-based detection method and adjust their movement accordingly.

This system ensures entities can move realistically and adapt to changes in the environment.

3.3. Detection System

Guards rely on a detection system that takes multiple factors into account:

- Distance between the player and guard
- Player noise level, which varies depending on movement state (sneaking, walking, sprinting)
- Relative position of the player (front, side, or behind the guard)

The detection system is based on a radius. The detection radius changes based on these factors above.

Additionally:

- Guards have a vision cone of 90 degrees.
- Detection is strongest from the front and weakest from the back.
- Detection values are visualized in-game for debugging.

This system makes guard behavior more nuanced and gives players a chance to avoid detection with careful movement.

3.4. Debugging and Visualization

To support testing and balancing, the game includes a debug interface with multiple visual tools:

- Pathfinding visualization: Shows the A* graph and the current path being followed.
- Obstacle detection: Highlights nearby obstacles.
- Flocking forces: Displays direction and strength of each force acting on sheep.
- Detection system: Visualizes guard vision cones and detection ranges.
- Entity info: Displays current state and data of entities in real-time.

There are also keybinds for controlling simulation flow (pause, step forward/backward) which are helpful for testing edge cases and observing behavior in detail. The debug mode can be active with the keybind shown in the game. All the options above can be toggled with switches inside the debug mode.

3.5. Sheep Flee Logic

The Fleeing force for the sheep is based on fuzzy logic that takes into account the noise the player makes and the position of the player. The fuzzy logic isn't triggered until the player is within a certain radius of the sheep. This mimics the limited eyesight of the sheep. The other antecedents, noise and Position, are not hard limits. These are based on the player's movement type and position relative to the sheep. For example, if the player is behind the sheep, it would be less likely to flee unless the player is moving loudly.

4. Reflection and Improvements

During the development of OutcastSurvival, we encountered several technical and design challenges. Although the core functionality of the game was successfully implemented, there are a number of areas that could be improved to make the game more efficient, realistic, and scalable in the future.

4.1. Improvements

There are a few things we could improve to make the game better. First, while guards use a state machine, they don't yet use pathfinding. Adding A* pathfinding to their behaviour would help them move around obstacles and follow the player more effectively. Right now, they don't really patrol either. Giving them patrol paths would make the world feel more alive and make stealth more interesting. Also some separation force between the guards. (Bevilacqua, Understanding Steering Behaviors: Movement Manager, 2013) (Bevilacqua, Understanding Steering Behaviors: Path Following, 2013) (Windesheim, sd) (Refactoring Guru, sd) (Buckland)

Finally, our state machines are written specifically for each character. Making the system more general would make it easier to reuse and scale in bigger projects.

4.2. Issues

One of the main issues we ran into was that movement was too twitchy. Guards would instantly go full speed, which looked unrealistic. We fixed this by adding small delays and acceleration so movement becomes smoother.

We also had problems generating the navigation graph using DFS. In some cases, it didn't cover the full map. Using BFS instead would be more reliable.

4.3. Future Ideas

If this game were to be developed further, we have several ideas that could expand both the gameplay and technical systems:

- Procedural map generation: Create random maps at runtime to increase replayability.
- Dynamic terrain: Include obstacles, cliffs, or areas that influence movement cost or visibility.
- Guard camps and loot zones: Introduce guarded areas with high-value loot to encourage risk-based exploration.
- More complex AI behaviours: Guards could communicate, call for backup, or respond to distractions.
- Day-night cycle: Detection values and vision could vary depending on time, adding more stealth strategy.

References

Bevilacqua, F. (2013, Febuary 15). *Understanding Steering Behaviors: Movement Manager*. Opgehaald van envantotuts+: https://code.tutsplus.com/understanding-steering-behaviors-movement-manager--gamedev-4278t

Bevilacqua, F. (2013, July 2). *Understanding Steering Behaviors: Path Following*. Opgehaald van envantotuts+: https://code.tutsplus.com/understanding-steering-behaviors-path-following--gamedev-8769t

Buckland, M. (sd). *Programming Game AI by Example*. Jones & Bartlett Learning.

Refactoring Guru. (sd). *Strategy*. Opgehaald van Refactoring Guru: https://refactoring.guru/design-patterns/strategy

Windesheim. (sd). *https://code.tutsplus.com/understanding-steering-behaviors-path-following--gamedev-8769t*. Opgehaald van Brightspace: https://leren.windesheim.nl/d2l/home/103607