AI for Games: Individual Project

## Git link: <https://github.com/rhodes167/AIProject>

# Introduction

This project involves the production of a technical demonstration for a simple stealth game system. The game involves a small surface upon which two actors can move around. One of these actors is controlled by the player, the other is controlled entirely by the AI systems implemented. Also on the surface, are numerous walls forming a simple maze. The actors will be required to navigate around these obstacles in order to interact.

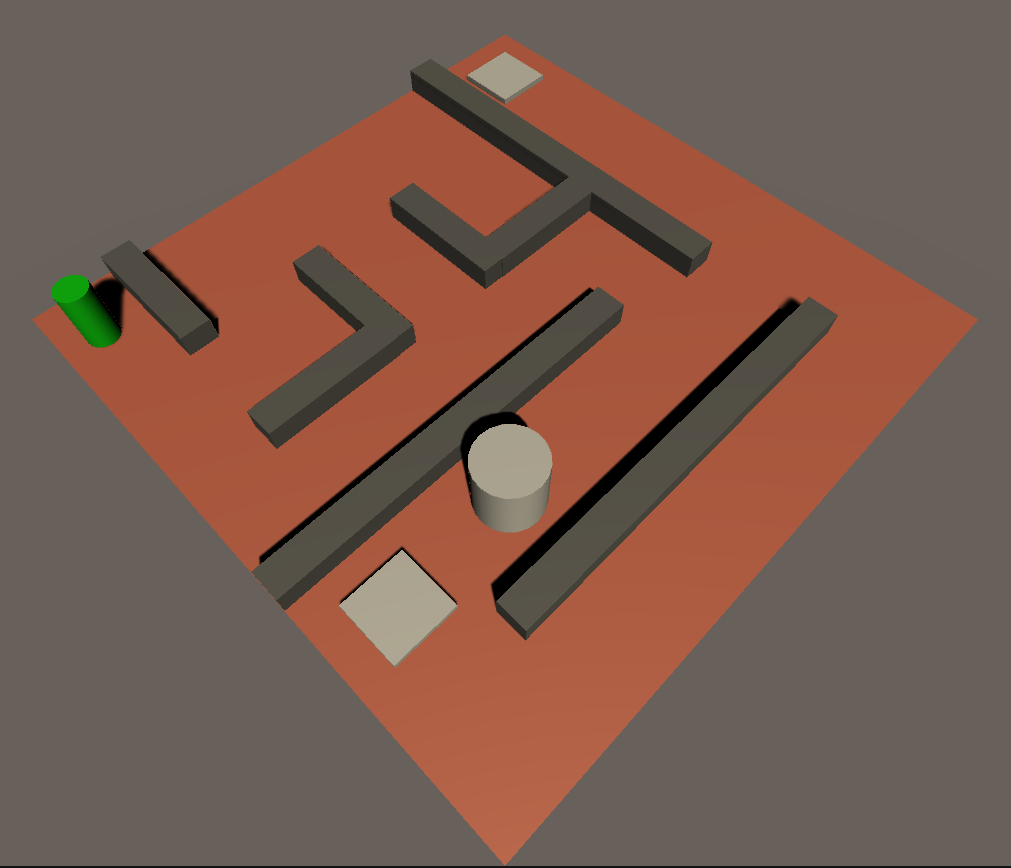
The player character must navigate these walls, and plot a course to a location selected by the player using the left mouse button. The other actor, filling the role of a guard, will patrol between two points in the maze. Upon reaching one of these points, the guard will wait before proceeding to patrol back towards the other location.

At any time during this patrol behaviour, if the player character gets close enough to the guard, the guard will become alerted. The alerted guard will then no longer follow the patrol route previously outlined, but instead pursue the player character. If the player manages to put enough distance between the player character and the guard, the guard will give up and cease chasing the player. At this point, the guard will return to its non-alerted state, and resume its previous patrol route.

A NavMesh was deemed necessary to ensure proper navigation around the wall obstacles. A NavMesh, also known as a Navigation Mesh, is a network of polygons used to designate traversable areas in an environment. They are used by AI agents to aid in pathfinding through complex spaces. While alternative methods exist, including the use of waypoints or grids, the ease of implementation of a navmesh in the chosen game engine (Unity), as well as the high level of readability. The clear visual representation provided in-engine allows for easy verification of valid implementation, as well as easy debugging should any issues arise.

The AI controlled guard character will feature behaviour determined by a Finite State Machine. A Finite State Machine (or FSM) is an abstract computation model in which an agent can exist only in one of a limited number of states. The states that define an FSM can contain the selection of simple actions and behaviours that the agent can exhibit. The agent can then respond to inputs and events by transitioning into another state. In games, these states would include different sets of actions an agent would undertake under certain circumstances. For example, a soldier may switch from a ‘shooting’ state to a ‘seeking cover’ state when it detects a live grenade nearby.

The choice to use a Finite State Machine was made due to the simplicity of the AI decision system. Finite State Machines are easy to implement and maintain, which lowers development time. In industry, this would also lead to lower development costs. FSMs are also easy to debug, due to the independent behaviours being easily identified, and the code being relatively easy to parse. The code naturally is split into self-contained sections which will never run simultaneously as agents can be only in one state at a time. With only a limited number of possible AI agent states; Patrolling, Idle and Chasing, the AI system for this project would not require any more advanced system. A more advanced system with additional functionality may incur increased development time and resources, and have a higher overhead while running.

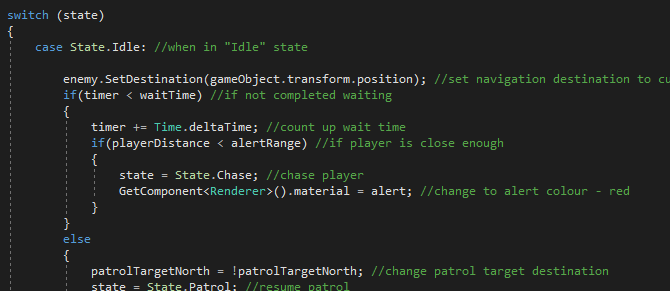


# Implementation Report

# The project was undertaken in the Unity game engine. This provided a good framework for a simple technical demonstration like this. The NavMesh was achieved through Unity’s NavMeshAgent AI system. This is a shortcut that was used so as to allow for greater focus on the primary AI system, the Finite State Machine. Diagram Description automatically generated

Implementation of the FSM allows for transition into the Chasing state at any time. However, transitioning out of the Chasing state will always return to the Patrolling state. This was decided upon to prevent idling in arbitrary positions on the game surface, and cause the guard to begin travelling to its previous destination immediately.

The Finite State Machine itself was structured around using a switch statement and an enumeration. This approach, while not the most expandable, is sufficient for a small project such as this.



Decisions regarding the changing of states was handled through if statements within the switch statement. While in larger projects, this would quickly become cumbersome, it was selected here due to the ease of implementation and limited scope of the project. There were also no checks undertaken to ensure appropriate behaviour upon changing FSM states. In industry, this is often included so as to avoid errors and aid in debugging. This was not included here as the triggers for changing state are based on easily determined variable comparison, for example distance between game objects. As such, it is unlikely to cause problems in this way.

Originally, a behaviour tree was considered as opposed to the finite state machine. This would have increased development time without any tangible benefit. All desired functionality for this project could be achieved easily through a simple FSM. The Finite State Machine approach also would be less intensive on system resources, allowing more instances of it to run concurrently. This would be beneficial if the project were to expand into a larger scale game featuring additional AI agents. It also benefits the project by allowing it to run more easily on low-end and legacy hardware.

# Discussion

The behaviour observed in the finished demonstration largely matches the desired outcome. Upon reaching close enough proximity to the player, the guard transitions correctly into the ‘Chasing’ state, changes colour, and begins pathfinding and moving towards the player.

Logo

Description automatically generated

The change of colour gives clear visual feedback to the player and informs them of this state transition, allowing them to respond appropriately. Once the player leaves the detection range, the guard actor transitions back into the patrolling state, once again accompanied by a colour change, and proceeds to its previously targeted destination node.

Logo

Description automatically generatedThe idle state for the guard is functional, though the positioning is not central on the destination pad.

This would not be an issue in a game developed beyond placeholder assets, as guard waypoints would likely not be visible objects. If necessary, this could be fixed by adjusting the trigger through which the guard actor transitions to the idle state.

The navmesh implementation itself works well for the most part. The player and guard actors can make their way through the maze without ‘cheating’ by cutting through walls, and both can navigate by finding the appropriate route in each circumstance. The guard actor occasionally can be seen slightly clipping into the walls, however this could easily be addressed by adjusting the boundaries of the navmesh used by the guard.

Logo

Description automatically generated

A separate navmesh could be considered for the guard actor’s patrolling state. It would more accurately mimic believable patrol behaviour if the actor remained more central on the path between the two destinations. This could be achieved by further increasing the navmesh boundaries. Cutting closer to the corners as it does in the current implementation would reflect the more urgent response a guard would have to an intruder, leading to a more believable game scenario for players.

The guard patrolling activity could be expanded upon in a full game to include full elaborate guard routines. This has been used in stealth games such as the Thief and Metal Gear series to add challenge by requiring the player to observe and learn these routines. It also adds to the believability of the game world, by further developing the enemy characters into more than just mechanical obstacles for the player to navigate past.

Further expansion in functionality could be easily added through the implementation of a vision cone for the Guard non-player character. This would allow for the player to traverse closer to the guard, as long as the player remains out of the guard’s line of sight, for example behind it.

An alternative implementation of the finite state machine would cause the guard to enter the idle state in the position at which the player character escapes. This could be used to better simulate the guard “searching” for the player. The decision not to act this way was to allow for a faster paced demonstration with a more prompt “reset” condition. This idle behaviour could be combined with the above suggestion of a vision cone by having the guard “look around”, further personifying the character.

Furthermore, the modification of the checks to ignore sightings through walls would allow for more engaging gameplay. The current implementation models the walls as “waist height”. While a common trope for video games, this was selected as a simplification consideration, as opposed to one that benefits the product. This was intended to reduce development time and debug complexity. One method through which a view obstruction system could be implemented would be a raycast from the guard character, projected in its line of sight. If it collides with a wall prior to the player, detection would not be triggered.

More bespoke pathfinding could have been used, as opposed to the current in-built Unity system. Examples for this could include the use of a bespoke A\* system to determine an optimal route across nodes. This was avoided to reduce development time and allow for focus to be put on the finite state machine component. The Unity implementation is also both effective and efficient for this project’s purpose.

Behaviour trees could have been used in place of the finite state machine. Finite state machines saw great popularity in early games and, in particular, those following the release of Valve’s Half-Life. Behaviour trees are much more popular in more recent games, approaching the de facto choice for AI behaviour. Behaviour trees allow for more flexibility, for example having the ability for an agent to be performing multiple tasks at once. Additionally, Behaviour trees are much easier to make changes to. The added flexibility is not necessary for the purposes of this project, however, and adds nothing to the functionality in this use case. For example, the ability of Behaviour Trees to run multiple branches in parallel provides no benefit to a basic system such as this, as it would not be utilised.

The choice of an FSM over Behaviour Trees lead to reduced development time due to it being an easier system to implement. The code behind FSMs is easily understood, making debugging much simpler. Finite State Machines also tend to have a very low overhead, allowing the program to run more easily on less powerful devices. In its current state, this game would not be particularly demanding even if it implemented higher overhead techniques. However, if the game were to be expanded with a far greater number of AI agents, this performance benefit would become cumulatively more significant.

# References

Technologies, U., 2021. *Unity - Manual: Unity User Manual 2020.3 (LTS)*. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Manual/index.html> [Accessed 2 June 2021].

Technologies, U., 2021. *Unity - Scripting API: AI.NavMeshAgent.SetDestination*. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.SetDestination.html> [Accessed 2 June 2021].