jamk

# AI In Games

## Workflow Report

Aapo Vanhainen

Learning assignment
AI In Games / Zsombor Faragó
Return date: 30.05.2025
Degree Programme: Business Information Technology

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

**Contents**



**Figures**

# 1 Introduction

For the final assignment of this course I wanted to create two distinct AI logics: A* pathfinding and Behavior tree. I spent upwards of 10 hours first searching and theory crafting how I would do these and started up working on the A* pathfinding first.

Ultimately, the time it took me to create feasible A* pathfinding, I ended up not having time to create proper behavior tree system, so ended up having simple state machine for the AI.

# 2 A* pathfinding

## 2.1 Beginnings

After I finished up with creating the basic algorithm for A*: f(n) = g(n) + h(n), I created node system and pathfinding system for them to communicate with each other. Then I started working on waypointgenerator, a script that handles the info and lays out map of waypoints in any level for a 3D game. Creating waypoints for flat surfaces was easy enough, it determines the gap between each node (grid spacing) and on which layers you want the waypoints on, aka where the AI can walk. This is usually Floor, Stairs or other walkable layers. Next, I wanted to add obstructions and how the waypointgenerator should avoid them when creating a path. Whether they are walls, props like tables or something else, they can be checked with Obstacle Layer. Also there is adjustable max height difference between waypoints, meaning if the value is 1, then if the difference between two waypoints in Y axis is greater than 1, it does not create the waypoint.

Now the waypoints are created in the level successfully, avoiding the obstacles and even out spacing for our choosing. But then came the stairs problem…

## 2.2 Difficulties

### 2.2.1 Stairs

One huge problem was inclusion of stairs. They are not obstacles, they are movable surfaces for the AI so they should access them easily, yes? Well for instance, right below the stairs the way-pointgenerator generated waypoints through the stairs on the surface and furthermore directly through the side of the stairs on to the stairs.

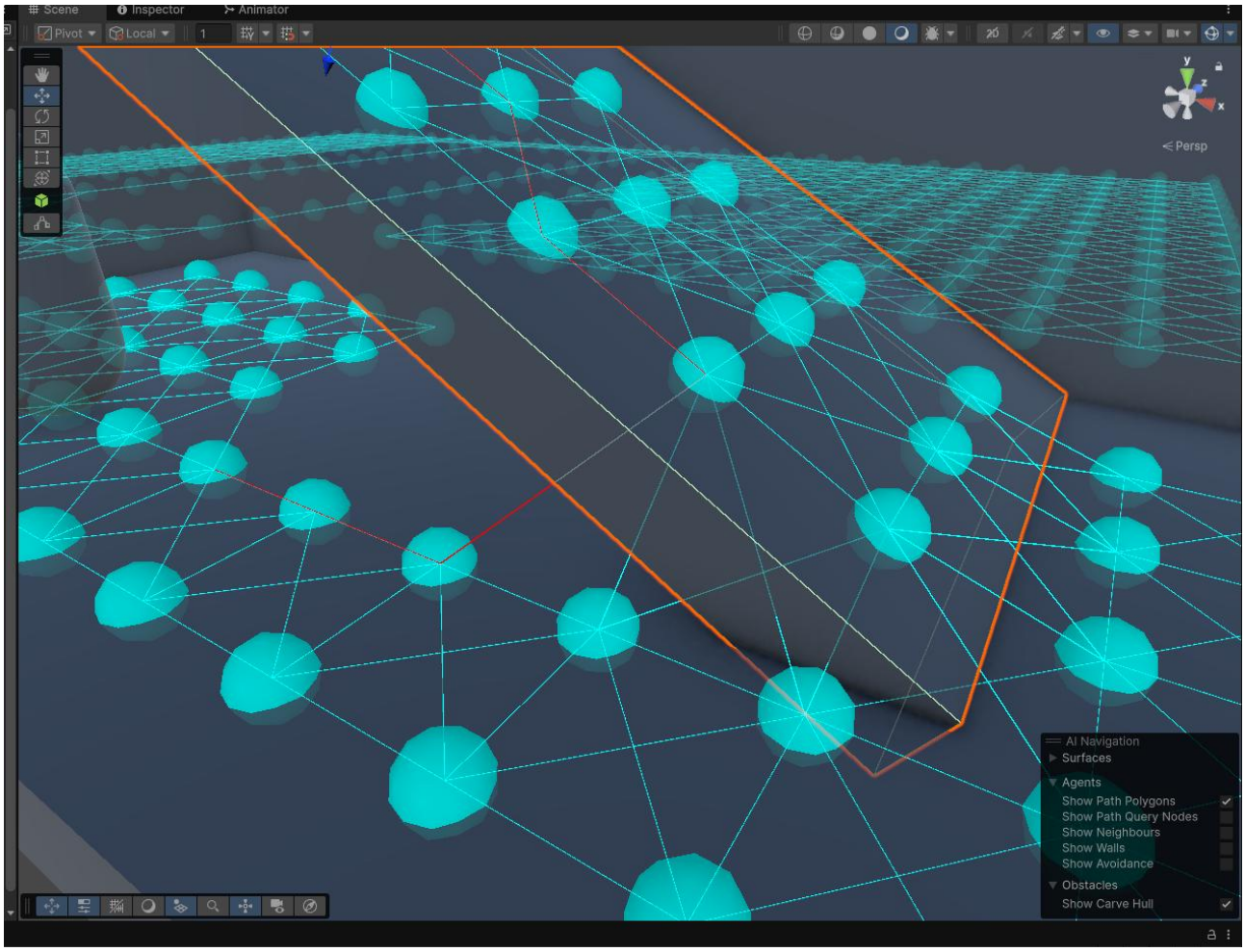


Figure 1: Waypoints going from the sides and behind, through the stairs on to the stairs.

First off, I wanted to fix the issue of having waypoints through the stairs. Another problem was creating waypoints on surfaces that the AI couldn't fit in the first place. I added raycast up from the waypoint of adjustable, 2 units, height to check if there are overhangs or anything else blocking the way. If there was, don't create waypoint. This fixed both problems very well. Next up was the sides of the stairs. And this opened a whole new world of problems.

### 2.2.2 Navigation from sides of the stairs

I tried multiple different approaches for this: having raycast spheres, boxes, rays etc. to check if there is another Floor layer next to the waypoint. In any cases it fixed something but broke something. What if the stairs are facing the other direction, now the X is on the left not in front? Then one stairway would be created with correct waypoints but the other wouldn't be. Spherecasts were difficult too since the waypoints were inside the floor or on the very surface of them. Okay, we up them a bit. Well now they are too high and don't help at all when we have waypoints very next to the stairs. Many similar occasions happened until I realized something. The enemy AI (and most characters in Unity) use capsule collider. If the check on the waypoint is the exact same capsule collider height and radius, then the AI can maneuver through the path, if it doesn't fit, then don't add the waypoint. This fixed basically everything. Having adjustable collider shaped check for waypoint generation meant that whatever your AI collider's size is, the waypointgenerator checks can it move here or not, then decides whether it creates the waypoint or not. For AI's that use box or spherecollider, this directly does not work but with little tweaks can be made possible.

This helped a lot in creating proper pathing for specific AI model but still it had problems with having nodes next to the stairs and allowing the AI to path through them. I tried multiple different approaches for this but nothing seemed to be a proper fix that wouldn't break something else in the system. First iteration of the AI I had them to move with transform.translate, then I wanted to add rigidbody movement for realism. The corners and small height differences on the sides of the stairs couldn't be done without either: transform through them by disabling collision briefly or making a jump with rigibody addforce. Making the enemy jump at corners just looked silly and going through collisions was even more stupid. What I ended up doing was adding character controller, which has step limit and step offset built in already. The reason I really wanted rigidbody movement was to add realistic ragdoll physics when the enemy dies but during this process I realized I should just have different prefabs for the alive and dead enemies, alive with character controller and dead with rigibody. Now that I added character controller, the small steps were no longer issue and the movement on the nodes worked perfectly.

# 3 State machine

I wanted to create behavior tree with complex enemy AI system but unfortunately there was not enough time to really invest into it. I ended up creating state machine for the enemy AI. The enemy now has four different states: Idle, Patrol, Investigate and Combat. Patrol state works with manually inserting empty game objects as patrol points, having the enemy AI either move them according to the list, top to bottom, or with toggleable feature, have the AI choose randomly the next point. Investigate was a bit harder but it works as follows: player has ability to make sound (currently just pressing F or toggling the Boolean) and if the AI is in the range of the sound, it will stop and find it's way to the sound to investigate. If there is nothing there, it will spin around, looking for the sound source, if it doesn't find player, then it returns to the patrol state and continues where it was left off. Whenever the AI sees the player (with adjustable vision range/cone), whether it is in Idle, Patrol or Investigate state, it starts combat state. In combat state the enemy tries to get to shooting distance of the player, then start shooting towards the player, while looking at the player all the time. If the player hides, the enemy goes to investigate the last seen position of the player and from there either continues combat or goes back to patrol depending on if it still sees the player.

Well, I ended up doing a bit more. The EnemyAI is now more robust with finding ideal paths not only to player but also to if player makes sound, it finds best possible path to a location where it can see AND can shoot the player from. It takes into account the enemy height and gun position in relation to where it should go. The numbers are fixed values for now but the gun position is reference to shootposition location, meaning it actually checks can the bullet fired from the gun reach the player. Further investigation how this works should be done when proper enemy model with animations are in place. Also normally when during Idle or Patrol if the enemy sees player, it moves to a position (usually slight adjust) where it can shoot the player, also looking from shootposition.

# 4 What EnemyAI.cs does?

## 4.1 Purpose

This script controls an AI enemy's behavior including:

- Patrolling along a route
- Investigating sounds and lost sight of player
- Switching into combat mode when the player is visible
- Using raycasts to determine line of sight and shooting feasibility
- Selecting strategic waypoints during investigations

## 4.2 State Machine Overview

The enemy operates under a state machine with four states:

enum EnemyState { Idle, Patrol, Investigate, Combat }

## 4.3 State Transitions:

- **Idle ➜ Patrol** (if no sound heard)
- **Any ➜ Investigate** (on sound detection or loss of visual contact)
- **Investigate ➜ Combat** (if the player is spotted)
- **Combat ➜ Investigate** (if sight is lost)
- **Investigate ➜ Patrol** (if search completes with no player found)

## 4.4 Core Methods Explained

### 4.4.1 Update()

- Calls HandleState() unless waiting for a strategic search
- Handles fire rate cooldown

### 4.4.2 LateUpdate()

- Moves along path (patrol/investigate)
- Rotates upper body toward player if in combat/investigate

### 4.4.3 HandleState()

Controls main decision-making:

- Transitions to **Combat** if player is seen
- If in **Combat**, checks if sight was lost and starts investigation

- Triggers investigation logic if sounds are heard
- Controls return to patrol if idle or finished scanning

#### 4.4.4 HandleCombatState()

- Aims and shoots at the player if line of sight and range are good
- Otherwise, recalculates a better shooting waypoint
- If sight is lost, transitions to Investigate with the last known position

#### 4.4.5 HandleInvestigateState()

- Starts scanning coroutine once enemy reaches target
- Resumes combat if player is seen again

#### 4.4.6 InvestigateRoutine()

- Waits at the last known position
- Rotates to scan for the player over investigateLookTime
- Ends with transition to Patrol if player remains hidden

#### 4.4.7 TryStrategicWaypointSearch()

- Coroutine to find a good vantage point using FindVisibleStrategicWaypoint()
- Falls back to FindClosestWaypointToSound() if no line of sight is found

#### 4.4.8 MoveAlongPath()

- Controls path-based movement
- Checks for investigation completion
- Prevents further movement when attacking
- Ends investigation properly by resetting flags and state

#### 4.4.9 FindShootingPosition()

- Finds a waypoint from which the gun can shoot the player
- Uses the shootPoint offset to simulate actual firing capability

#### 4.4.10 CanSeePlayer() / HasClearShot() / CanShootPlayer()

- Line-of-sight checks using raycasts or spherecasts

- Considers angles and obstructions for visual logic

#### 4.4.11 Patrol()

- Chooses a patrol point (random or sequential)
- Finds a path to that patrol point

#### 4.4.12 AimEyesAtPlayer() / RotateTowards()

- Handles body and upper body orientation
- Keeps enemy looking toward the player smoothly

### 4.5 Highlights of Improvements Implemented

- Proper separation between **vision from eyes** and **firing from gun** using simulated positions
- Strategic investigation uses cover-based logic with fallback
- Transition fixes to prevent reverting to Patrol too early after losing sight
- Improved realism and intelligence during investigation and attack

# 5 What Waypoint.cs, Pathfinding.cs and WaypointGenerator.cs do?

## 5.1 Waypoint.cs

This script defines a basic **graph node** used for AI pathfinding.

**Key elements:**

- Position: World-space position of the waypoint.
- Neighbors: Other connected waypoints (edges of the graph).
- A* pathfinding fields: gCost, hCost, FCost, and parent are used to calculate the shortest path.
- Constructor initializes the waypoint's position.

This is a lightweight data container used by the pathfinding system.

## 5.2 Pathfinding.cs

A static class that implements **A* pathfinding** on a graph of Waypoints.

**Key method:**

- FindPath(List<Waypoint>, Vector3 start, Vector3 end):
  - Finds the shortest path from the closest waypoint near start to the one near end.
  - Uses A* logic: maintains an open set and a closed set, evaluates FCost = gCost + hCost, and uses parents to trace the path.

**Supporting methods:**

- GetClosestWaypoint: Finds the waypoint closest to a given position.
- RetracePath: Backtracks from the end waypoint to the start to build the path list.

This system is used to move the enemy along calculated paths, avoiding obstacles and using clear terrain.

## 5.3 WaypointGenerator.cs

Generates a grid of waypoints across walkable surfaces in a defined area, checks for clearance, and connects them.

**Key Components:**

- **Waypoint generation**:
  - Casts rays down in a grid pattern over a scan volume.
  - Checks for floor collision and ensures there's enough vertical space (using Physics.CheckCapsule).
  - Prevents placement of waypoints too close to each other.
- **Waypoint connection**:
  - Connects nearby waypoints (in horizontal and vertical thresholds).
  - Ensures there’s no line-of-sight obstruction (via raycast).
- **Debug gizmos**:
  - Shows spheres, capsule clearances, and neighbor lines in the scene view for visualization.

This script builds the navigation mesh that the enemy AI uses to understand traversable areas and compute paths.