COMP-1864 Report

Kirk Hogden 001115381

Content Page

Introduction

A proposal was made for a game where the level is different every time the game is loaded, and AI can still navigate their way across it without challenge. This challenge was chosen because technologies such as Unity have limited tools to suffice this objective. The built-in AI navigation package is instead designed to bake a mesh around what hand-made levels the user has already developed for agents to move across, but this project is to use computer-generated levels. (Unity Technologies, 2023)

The AI that takes place in this project serves multiple purposes. One purpose is to increase productivity of the project. By having a computer assist in development, it can achieve visions that would be almost impossible if attempted without.

Another purpose is interactivity. Including NPCs into the game allows simulating a form of social interaction without the need of a second player, giving a realistic experience to gameplay. This is exaggerated when NPCs can find their way across levels with ease and are aware of the presence of human players. The need for NPCs grows more as games become more advance, which encourages this project to push the capabilities of making the AI smart. (Jagdale, 2021)

This report reflects on the development of the project, exploring the forms of artificial intelligence that took place. It'll explain how AI navigation works, and how computer-generated levels were implemented. A conclusion will discuss final thoughts on what went well and what could've gone better.

Procedural Generation

To make sure the level is almost never the same as the last time it was loaded, Perlin Noise is used to generate a grid-like level into use. A procedurally generated level gives advantage of making a level to a desirable size quickly without the need to developing it by hand. Perlin was the chosen algorithm over other noise types because it takes the form of an arbitrary grid, in which the game's level takes the form of a grid built up of tiles.

(Archer, 2011)

Having a level that's different every time would mean an AI navigation system would have to be self-made rather than using built-in game engine tools or third-party assets.

To implement procedural generation into the project, a 'LevelGrid' class was created which has the role of using inputted settings to generate data for a level. This doesn't instantiate game objects into the scene but simply creates readable data on the Unity inspector. Doing this can allow for opportunities to make a save mechanic so that the application can remember how a level was last left before it was closed. The Unity inspector allows the user to customise how generation works, such as the X or Z axis of the level widths or input a seed to work with the Perlin noise generation mechanics. A list of tiles with a possibility of generating in the level grid is included, where the user can input what tiles that they want to be added, and the chance value they have of filling a particular coordinated grid based on it's Perlin noise gradient value. Separate from the list of generatable tiles, there's a similar list for tiles that are specified to generate always only one time. This makes sure that one of each spawner type is generated into the grid. A spacing option was included in the inspector for how far out the tiles will be from each other, however it doesn't have full support and will break the rest of the game mechanics if tampered with.

The seed input is the most important generation setting of the LevelGrid class, and because of this, a seed input field was included in the main menu of the application. This allows the user to have some input in deciding how they want the level to appear, while at the same time letting the computer take over the heavy work. If the player has no desire for inputting a seed however, they may leave the field blank and the application instead will choose for itself. The seed value has an impact on how the level is generated because it is multiplied within the Perlin noise values that decide what tile type should fill in each grid on the level.

A ‘LevelManager’ class has the capability of taking data from ‘LevelGrid’ classes and instantiating objects into the scene based on what it reads. A function for placing down tiles is included, which also sets up a node above it to assist AI navigation if the specified tile is walkable. This function would’ve been useful if level-editing mechanics were included in-game. Using an IEnumerator, the level instantiation phase allows the player to watch the level get built tile after tile for a satisfying feeling.

A ’Tile’ scriptable object class was created, allowing for creating tile data in the assets folder. This class would contain fields for specifying the tile name, what prefab it uses when being generated, and if it’s walkable. If it is walkable, then a node game object will be instantiated above the prefab when this tile is generated into the level.

The LevelManager class collaborates with a ‘TileManager’ class. This manager acts as a database for all the useable tiles in the game. A method for accessing each tile by its given name in the ‘Tile’ scriptable object is included so that they can be found in easier ways rather than just having to know it’s index number in the list.

Interactive Intelligence

Walkable tiles had nodes above them which had scripts. These node scripts contained data such as neighbour nodes on all four directions. AI would use information from these classes to see which neighbour node is closest to their destination and move towards it. While it allowed some form of intelligence, it would situations such as the AI getting stuck in dead ends or going back and forth when unwalkable tiles were placed in the way.

Dijkstra’s algorithm was initially planned to be implemented but proved to be difficult. It would also be very expensive in performance as it tends to scan areas which aren’t as certain to find a clear path. This algorithm was desired because it scans every part of the level, and while it may look even in less likely areas, this would be suitable in a level that’s different every time it’s generated. (Garg & Devi, 2023)

Finite state machines were applied to AI to have them behave differently depending on their situation. AI will have a moment of patrolling, where they will move around in nodes near to their home spawn. After a short period, they’ll engage in a provoke state. How each AI behaves when provoked is different. An unused flee state has the AI run away from the player, moving the opposite direction from them. The flee state would’ve been useful in situations such as when the player gains abilities that allow them to overpower the NPCs.

To make different AI behave differently from each other, child classes were made deriving from a parent ‘AiBehavior’ class. While each AI had the same abilities, their movement would differ. Below is a list of the different NPCs included.

NPC Behaviours/Types

The ‘AngryBlocky’ when provoked will set its destination to be the player’s position, having the AI attempt in colliding with the player. This can allow opportunities for mechanics such as damaging the player when touching the NPC, which would make this NPC the most aggressive as it’s trying to get to the player. The ‘AdoringBlocky’ is similar, however they’ll aim for the tile in front of the player when provoked. This allows for manoeuvres to trick the AI if understood, however can still be seen as the second most aggressive if not careful when moving forward.

‘ShadowBlocky’ works alongside the ‘AngryBlocky’ when provoked, acting as their shadow. They’ll move towards a tile that’s four meters in front of its partner and four meters to the right. If the player gets too close to this NPC however, it’ll instead head straight for the player’s position. This gives off ambushing tactics between the ‘ShadowBlocky’ and ‘AngryBlocky’ to capture the player together.

Lastly, the ‘MilkyBlocky’ is the least dangerous when provoked. It will locate any moving milk bottle across the generated level and move towards it. If the milk bottles were pick ups for the player to collect, this specific NPC could be seen as defending them to prevent the player getting them first. Coding supports for this NPC to find a new milk bottle if the one they were initially focused on gets destroyed.

An additional game object that uses NPC programming, although not necessarily an NPC is the prior mentioned milk bottle. It shares the ‘AngryBlocky’ AI behaviour child class, although with a reduced provoked time so that it remains in the patrolling state for most of the time. Allowing the milk bottle to move around makes for a challenge in the player having to chase it across the level.

Conclusion

Although Dijkstra's algorithm wasn't added, the AI show a level of intelligence to at least know where their destination is and use programmed nodes to know where they can and cannot move across. This is an achievement in the case scenario that the level is always going to be different rather than using a baked-mesh on a hand-made level. This could be something to explore and attempt again at implementing if coming back to the project in the future.

If a pathfinding algorithm was implemented into the project rather than simply having the AI see what node is closer to its destination, they'd prove more efficient within the project. Finding an algorithm better than Dijkstra's for the specific project may benefit the performance. When reviewing A* on pathfinding visualisers, it would seem that perhaps using that algorithm would instead have suited the project more so long as it's friendly towards levels that constantly change.

Including more interactable features in the game could allow opportunities to make the AI more advanced. Because of the lack of features the game has, the most that AI can do is navigate their way across the level and hunt down the player in their own ways. An openable door for example could allow for AI to be interested in wanting to open it or find consumables around the level if a health system was included.

Rather than having the NPCs toggle between being provoked and patrolling around their homes, it might suit better to have them be provoked in different ways such as if the player were to get too close to them. Toggling between the two states over a period was chosen instead because the patrol state was to be seen more as a point where the player has a break from being chased across the level. An idea that could make the NPCs seem more realistic would have been to use Raycasts to act as their eye sight, in which they'll become aware of the player's presence when that line hits into their collision.

An intelligence/difficulty level would be useful for the AI too, which could impact their reaction times towards performing tasks. If the player could choose the AI difficulty, this could allow the player to give themselves a challenge that they find desirable to their skill level.

A problem that occurred when developing the level generation mechanics was the lengthy steps just to add a new tile into the project every time. A scriptable object had to be created for each tile which contained the tile properties, which would then have to be assigned to the TileManager tiles list so that it could be used by the LevelManager. Prefabs would then have to be made for these tiles which had to be assigned into the tile scriptable object item. Such lengths to create an individual tile type isn't friendly in the productivity of the project and very time consuming.

The overall accomplishment of this project was that a form of AI other than using built-in tools was created. AI navigation is a difficult task to complete which makes this project an achievement, and there will always be room to come back and improve it in future.

References

Archer, T. (2011). *Procedurally Generating Terrain.* Sioux City, Iowa: [no publisher].

Garg, S., & Devi, B. (2023). Shortest Path Finding using Modified Dijkstra's algorithm with Adaptive Penalty Function. *International Conference on Computing and Networking Technology (ICCNT)* (pp. 1-9). Delhi, India: IEEE.

González-Calero, P. A., & Gómez-Martín, M. A. (2011). *Artificial Intelligence for Computer Games.* New York, USA: Springer.

Jagdale, D. (2021). Finite State Machine in Game Development. *International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)*.

Unity Technologies. (2023). *AI Navigation*. Retrieved from Unity Manual: https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/