Kernmodule 3 AI

Celine de Wijs - 3013898

Introduction
My game is about being a cat and just wanting to do cat stuff, without being bothered by a child. The game has no win or lose conditions, but is meant to be enjoyed for as long as the player wants. As the cat, the player has to walk around and avoid the child. Fortunately there's an ally called Parent, which will try to keep the child from running after the player.

A* Pathfinding

How the A* algorithm works
The A* algorithm tries to find the shortest path between node A and B. For that, it first needs to find out how far apart both nodes are.

The algorithm starts by looking at the surrounding nodes of the starting node, adding them to a "open" list and calculating the following values for each of them:

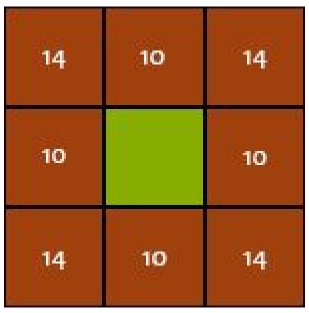
- G_cost = distance from starting node
- H_cost = distance from target node
- F_cost = G_cost + H_cost

It's going to look which of the neighbours has the lowest F_cost (including the current node) or if a neighbour has the same F_cost, but a lower H_cost as the current node. That node will be chosen as the next current node.

The current node will be "closed" so it won't be evaluated anymore and its neighbours will be added to the "open" list.

The algorithm will loop this progress until it has reached the target node.

How I implemented it
To get the distance between neighbour nodes, I give the horizontal and vertical nodes a distance of 1 and the diagonal nodes a distance of 1,4 (square root of 2). To make calculations easier and not bother with floating point values, I multiply it by 10.



I made three classes called “Node”, “Grid”, “PathFinding”, and “PathRequestManager”.

- The Node class defines what a node is. It knows if it’s walkable or not, what its world position is, what its point on the grid is, what its parent is, and what its A* costs are. The F_cost is declared as a struct, because you only want to get the value and not set it. The class has a constructor that defines all these variables.

- The Grid class defines what the grid looks like. It asks what Layer Mask it should use to define its obstacles, it contains the size of the grid in the world, know what size nodes are and has a list of the nodes on the Grid called grid.

- The Pathfinding class handles the algorithm. It knows what the position of the seeker and the target is. It has a function called “FindPath” which it loops until the path has been found. Then it uses a function called “RetracePath” to get a list of nodes of the path, starting with the start node. Lastly, it has a function called “GetDistance” which is used by FindPath to calculate the distance between the current node and the neighbour (as I explained here above), but also to calculate the H_cost of the neighbour node it’s looking at.
- The PathRequestManager is the class that the AI’s will communicate with to request a path. In its turn it communicates with the Pathfinding class. The PathRequestManager makes use of a Queue in which it stores the requests it got by an AI. In a static function called RequestPath it receives requests from the AI and adds it to the queue. Then it starts processing it with the function “TryProcessingNext”. This function is the one that calls “FindPath” from the Pathfinding class. Then the pathfinding class calls “FinishedProcessingPath” from the PathRequestManager which gives the path back to the AI.

Behaviour Trees with Panda BT

What is Panda BT
Panda BT is a scripting framework based on Behaviour Tree for Unity. By writing BT scripts you can define the behaviour of a GameObject. A Behaviour Tree is a hierarchy of nodes, where the deepest nodes are tasks. These are implemented in C# and invoked by the BT scripts.

Why Panda BT
I chose this tool, because I was able to create trees in a way that’s very similar to writing code. It also looked clutter-free and simple to use. After some tests, I was very happy with how easy it seemed to create behaviour trees.

Implementation
For this project I have created two AI’s called “Child” and “Parent”.

- Child is minding its own business and playing by jumping up and down. However, once it has eyes on the cat (the player), all it wants to do is catch it.
- Parent is guarding the place and especially the child. Once it notices the child has left its playing spot, it goes after it to bring it back. Then it will continue patrouillering.

Child
The Child has four states: Idle, ChaseTarget, CaughtPlayer, and Play.
In its Idle state it will check if the player is close and simultaneously fire the Play state. Once it has found the player, it will go to the ChaseTarget state, which runs if the player is near and the child isn't captured by the parent. It will request a path to the player and simultaneously check if it has already reached its target.
If it catches the player, it will hold it for three seconds and then release. Then it will go back to its starting position and returns to Idle.

Parent
The Parent also has four states: Idle, ChaseTarget, CaughtChild, and Guard.
In its Idle state, it will check if the child has left its starting position and simultaneously guard the place by walking around the field. Once the child leaves its position, it will go to the ChaseTarget state in which it will get the position of the child and goes after it. Once it catches the child, it brings it back to the child's starting position. Then it will go back to its Idle state.