Matthew Roy

EGP-410 - AI For Games - Final Project with Nick Robbins

[Matt]

Architecture in this project is based off of Nick's assignment 3, my assignment four A* and uses the Steering system from DeanLib. Within new kinds of steering, State Machine helps switch between the kinds of pathing. Two of those states also performs A* across two levels of a hierarchical grid.

Grid, Nodes, and Connections [Nick]

When we started designing the game, we decided to change how the graph worked in comparison to the code from the pathfinder code from Deanlib. Matt and I came up with the idea that the nodes would be merely the intersections within different blocks within a city, and then a sewer. That way there would be less nodes to deal with and the units can path via the streets to each intersection. With this design in mind however, the code for how the graph works had to completely change as well. So I completely built the graph, node, and connections all from scratch to work with the game, this also included loading in the nodes and connections from textfile. It works by loading in all the nodes from the nodes textfile which includes which level the node is on, its ID, and its position. After that is completed, the graph then loads the connections, stringing the nodes together as is described from the connections file. For the ai enemies, in their steerings, if the connections that they are going towards is on another level than they are, then they will transfer to that new level. For the player, there are door objects which check for the player to move it to the other map.

The issue with the way that I set up the graph though is that it seems very flexible because of its text based nature, but it is very conformed for this specific game itself. Because of this fact, when it comes to the debugger, there isn't any node manipulation in it because before we knew it the way that the graph was set up was too entrenched in our codebase and manipulating the nodes further would probably break the game.

Debugger [Nick]

The debugger is obviously not as far as it probably should be, but that's because of the error on my part on how the codebase was constructed around the graph as it was designed. We created the grid with a specific setup in mind, and it made it impossible to create further flexibility. I think that it was also an issue of time, because we had chosen not to use dean's pathfinder code in substitute for our own, it made the understanding of the codebase better, but

removed possible flexibility with the code if we had stuck with the original codebase. If I had more time I would've put in node creation and deletion with the ability to string connections between them, but there wasn't enough time for that.

What it can do though, is allow the player to put in more power ups and money drops on the screen. At least that's something right? The stuff I had shown off in the demonstration beforehand where I was adding and removing nodes were removing them, but mainly from the rendering, and their connections were still stored in the map within the graph so objects could still pathfind.

Keeping Track of Objects on Levels[Nick]

Just a side note for the way that the system keeps track of objects over levels. As stated before, the nodes are loaded in with their levels. For all the other objects in the game, they are created with a level. When they are called to be rendered, if their level isn't the same as the current state which is stored in Game, then it isn't rendered. This is the same with hit detection with units and walls on the two levels. If they are not on the same level, then it isn't considered. The objects steering behaviors still do happen however even when not being rendered.

PoliceSteering [Matt]

PoliceSteering is a child of Steering that controls the behavior of the enemies in the game. It uses a state machine that contains a current state that it performs each frame and returns. Its states include Wander, Chase, Flee, and Dead. More on these specific behaviors below.

DeadSteering [Matt]

DeadSteering is a simple steering algorithm that does nothing but count up a timer to tell the enemy's state to change and "respawn" on the map.

WanderToNode [Nick]

The wander to node is a wandering behavior built with the grid setup in mind. It works by choosing a target node, pathing to it, and then choosing one of the connections that

node has to path to the node that it leads to. It allows the AI object to path randomly around the grid and even show up on the other maps as well.

PoliceSeek[Nick]

For this behavior, this uses A* to path from the AI's current posision to where the player was last seen. In the kinematic unit, each object keeps track of whether they have come across a node and it updates it as the last node they were on. So this algorithm generates a stack using Matt's A* code to path to the player through the last node it was just on. While the stack has objects, it will follow the stack until its empty where it will use the A* to search again. This code should work across the maps as well, so the following enemies should appear in the other map as well.

PoliceFlee[Nick]

For the flee it's simple like the wander code. The object wants to get as far away from the player as possible, so when the player gets a power up, the enemy will run away to the node that is the furthest away from the player's position. So it gets the connections of the node that it is on now and will calculate the distance between the options and the player's position, and it will choose the node with the furthest distance from the player.

PlayerSteering

State Machine [Matt]

My StateMachine class contains a list of states, a list of connections, and the current state. Each state is a steering object so that they can easily create movement behaviors. Nodes are added by passing in pointers to said node while connections are created using a function that takes pointers to the nodes that are "to" and "from" in the connection. A steering object with a State Machine can call doCurrentState to perform the current state. ChangeCurrentState is called to change the current state but only if

connectionExists returns true that a connection between the currentState and the node passed into changeCurrentState exists.

Circle Collision [Matt]

Circle Collision is a class that contains many collision algorithms pertaining to circles. It contains three functions that do circle on box collision with the different types of units (walls, water, and doors). I had trouble getting the constructor to work properly in this class so each collision function has a bool that keeps track of whether or not things have been initialized or not (basically if the class has been able to fill its list of each type of unit with a “BoxWithCenter” struct that contains a pointer to the object and a newly calculated center point). Each function also has a bool that keeps track of whether or not a collision has occurred to return once its done checking because the functions were breaking and returning prematurely. Circle Collision also contains a simple circle on circle function which is used for collision between the player and enemies as well as player on pickups.

Raycast [Matt]

Raycast is like Circle Collision in that it is a class that contains a collision algorithm. In the end, I wasn’t able to get Raycast to work so we adjusted the gameplay accordingly. Instead of the enemies having to see the player to begin pathing to them, enemies will path to the last node the player has visited (which I think makes for some interesting gameplay).

Pickup Unit [Matt]

PickupUnit is very similar to WallUnit in that they are both children of KinematicUnit with a few extra values. While WallUnit contains four Vector2Ds referring

to each corner, PickupUnit simply contains a radius to help calculate circle on circle collision with the player.

Audio Manager [Matt]

Like many of the other managers, the audio manager has things added to it in the initialization of game to be stored for later. It contains a list of sound effects and an instance of music. The music begins playing as soon as its added to the manager while the sound effects can be called and played using a string ID whenever needed.

Final Thoughts[Nick]

One of the things I noticed on the project was that as time went on, more and more objects got more entangled with the graph. Things needed to know about the graph, connections, and nodes just to get a position of a node which came up very commonly. The codebase seemed to be slowly getting more and more connected with itself which ultimately created the problems that we faced now. Finally, the code does have some memory leaks which we have accepted as a fact of sacrilege when it comes to programming. We know that the leaks are coming from somewhere in the A* pathinding. I tried clearing out the node records and making sure anything that is being constructed with “new” was to be deleted, but it was still generating a leak which we are unsure of. We are sorry for the state the game came out it. It was mainly a great overtaking on our part for what we needed to do with the time we had left along with all the other projects and stuff we had to do. If we were to do it again, I would’ve just stuck with Dean’s code and tried to use it and work around its issues. That way, the tilegraph would have been much easier to use both when it came to the pathfinding and the debugger functionality.