Legends of Arborea

Thomas Meijers, Markus Pfundstein

March 2015

1 Introduction

For the third assignment of the course *'Datastructuren'*, we have chosen to implement the game Legends Of Arborea. Our technology stack was plain Java8 in combination with the Slick2D Graphics Engine.[1] We have chosen for an object-orientated programming approach, that allowed us to quickly prototype a small game engine that can easily be customised and extended.

2 Data-structures

The data structures used for this game are quite simple. Most consist of arraylists or arrays, while the implementation of the A*-search algorithm uses a single linkedlist to keep track of active paths. There was no need for structures like queues for things like animations since we used separate threads to handle updating, rendering and AI thinking. The arraylists are used because in the game code these are always of variable length with each new action, while animations are held in a four dimensional array which has a static size (defined in LegendsOfArborea).

2.1 Game engine

The current underlying game engine is easily expanded with more types of units, tiles and different sized maps (up to 19 x 19 x 19). All underlying methods have been built so that they can handle units with different stats then the standard soldiers and generals (one could for example implement bowmen which have an attack range greater than 1, or thieves which can move two tiles). The same is true for different types of tiles, tiles could harness bonuses or penalties to attack range and weapon skill. Even the implementation of new textures/animations requires only the adjustment of a few static variables.

[1] `http://slick.ninjacave.com`

3 AI

Because the rendering and game logic gets evaluated and calculated in the main thread, the computations for the AI are done in a second thread. Whenever the AI has to think about the next set of moves, a new thread starts up and calculates a move for each unit. All moves get appended to a list, which then gets pushed onto a queue that is synchronised with the main thread in order to avoid race conditions. When all moves are pushed onto the queue, the AI thread terminates. On the main thread, every move gets popped from the queue and then executed in the main loop. When the queue is empty, the AI is done and the control is given back to the player.

3.1 Random AI

In the current implementation, we have implemented two different AI's. The first is a Random AI that moves and attacks completely randomly. This was mostly done for testing purposes of the AI Engine as not much programming was needed. It goes without saying that it is very boring to play against this AI.

3.2 Aggressive AI with A* Path-finding

This implementation is a rule based AI that uses A* to find the tile to move to. To start up to 5 targets are selected based on low hp and shortest distance to unit belonging to the AI. For each target, it then finds a suitable attacker, this time based on the unit that is the closest. It then uses A* to find a suitable next tile (in the rare case no path is found (one or both units are blocked) it continues with the next target) and moves the unit. If the target is encountered, the AI attacks. It then looks at how many empty spaces the attacker will have surrounding it and tries to move as many 'supporting' units next to the attacker as possible and when these units can attack they will attack the target with the lowest HP. If at one point the AI is out of units, it will break out of the loop and return all moves, the other case is when all targets handled, then the AI moves it's remaining units as close to the attacking/supporting units as possible and attacks whenever possible. All pathfinding is done with A*, and generating all the moves only takes a fraction of a second.

3.2.1 A*

The reason A* is used is because it can easily find the best path while having the lowest complexity out of other search algorithms which are easy to implement. A search algorithm seems overkill but since units can be blocked and we do not want the AI to make the greedy best move and attack a random target we need it.

3.3 Implementing a new AI

The current layout of the code makes it very easy to add a new AI. All the developer has to do is to create a new class that inherits from the class *AI*. In the newly created class, the programmer can simply overwrite the method *doThinking*. In this method, the programmer can implement all AI logic engine and the game itself will handle the rest. This highly object-orientated architecture makes it great for trying out new implementations of AI techniques.

4 Discussion

The assignment itself went very well and we are very satisfied with the result. It would be very interesting to see what is possible with the game if someone would take the time to develop a proper AI for it. Also more units and different terrain modifiers would greatly extend the strategic component of the game. The engine that we have written is easily to extend with new functionality. It can therefore readily be used for future research and extensions.