Otto – Toteutusdokumentti

Ohjelman rakenne:

Ohjelman päälogiikasta on vastuussa worldmap.py joka sisältää kaikki kartan generointia varten vaadittavat algoritmit ja logiikan reitinhakua lukuun ottamatta. Erikseen oleva tiedosto astar.py, joka on omassa implementaatiossani riippuvainen worldmap.py:stä, taas sisältää A* -reitinhakualgoritmin ja siihen vaadittavan logiikan.

Astar.py käyttää kahta itsetehtyä tietorakennetta, NeoSet (jonka tarkoitus on jäljitellä Java-tyylistä ArrayListiä) ja NeoQueue (joka on yksinkertaistettu versio Pythonin omasta enque-luokasta joka on aika monimutkainen jonorakenne). Kumpikaan ei ole riippuvainen mistään muusta luokasta-

Itse ohjelma ajetaan otto.py –pääluokasta, joka sisältää myös vaadittavat renderointi- ja syötteenkäsittelyfunktiot (muttei mitään varsinaista ohjelmalogiikkaa).

Riippuvuudet:

Pääluokka otto.py on riippuvainen PyGamesta, joka sisältää vaadittavat toiminnot ikkunoiden luomiseen, syötteen lukemiseen sekä ohjelman graafiseen renderoimiseen. NeoSet.py taas on riippuvainen NumPy-paketista, jota käytetään koska itse Python ei luonnostaan tarjoa tukea primitiivisiin array-rakenteisiin jota käytin NeoSet:iä luodessani.

Suorituskyky ja O-analyysi:

Itse kartan luonti tapahtuu ajassa O(n), jossa n on maksimissaan koko kartan koko + metsien koko. N. 64x64 –kartalla ja ~1200 metsätilellä ohjelman käynnistämiseen ja kartan luomiseen menee n. 0.87 sekuntia omalla koneellani, ja jopa moninkertaistaessa koon (esim 128x128 –kartta ja ~12000 metsätileä), aika pysyy käytännössä samana (n. 0.88 sekuntia): itse kartan luonti on siis tehokasta, mutta reitinhakualgoritmi kärsii suuresti kun kartan kokoa muuttaa.

A* -algoritmin käyttämät tietorakenteet NeoSet ja NeoQueue ovat periaatteessa yksinkertaiset: NeoSet on vain perinteinen järjestämätön array joka muistaa indexinsä (ensimmäisen vapaan paikan) jonka koko tuplataan (eli uusi luodaan ja vanha kopioidaan päälle) kun se täyttyy. Sen muistivaativuus on siis O(n) jossa n on enintään kaksi kertaa syötteen määrä; myös siitä etsiminen tapahtuu lineaarisessa ajassa O(n), ja siihen lukeminen ja siihen lisääminen on vain yksinkertainen O(1) – tosin jälkimmäisessä tapauksessa jos array täyttyy ja sen koko täytyy tuplata, lisäämisoperaatio on kompleksisuutta O(n).

NeoQueue on yksinkertainen jonorakenne joka periaatteessa pohjautuu Pythonin dequeue-luokkaan. A*-algoritmi tarvitsee jonorakenteen toimiakseen tehokkaasti (kts. Priority queue), ja ainoat operaatiot mitä A* tarvitsee jonolta, enqueue ja dequeue, tapahtuvatkin ajassa O(1).

A* on loppujen lopuksi vain hiukan loogisempi leveyssuuntainen haku, ja ei voida vannoa että sen käyttämä heurestiikka oikeasti auttaa. Pahimmassa tapauksessa sen aikavaativuus on O(n^2), tosin käytännössä ei ikinä jouduta tuohon tilanteeseen.

Osittain tehottoman NeoSet-rakenteen takia omassa implentaatiossani A*:lla menee 64x64 –kokoisella kartalla jopa n. 9 sekuntia löytää reitti kahden satunnaisen pisteen välille. Vertailun vuoksi samalla algoritmilla kestää vain ~0.34s(!) jos käyttää Pythonin sisäänrakennettua set() –tietorakkennetta, joka perustuu erittäin optimoituihin hajautustauluihin. Siinä mielessä NeoSetin suorituskyky on kaamea tähän tarkoitukseen, todennäköisesti etsimisen hitauden takia: O(n) vs hajautustaulukon O(1) on valtava ero.

Puutteet ja parannusehdotukset:

Loppujen lopuksi vain murto-osa määrittelydokumentin ideoista tuli toteutettua. Järvien ja jokien teko jäi puoli-implementoiduksi (ja sitten poistetuksi) lähinnä algoritmin hitauden takia: jokien vetäminen A* -algoritmilla olisi ollut uskomattoman hidasta huonon NeoSet-tietorakenteen vuoksi.

Jos olisi ollut enemmän aikaa, olisin korvannut koko NeoSetin jonkinlaisella hajautustaulutoteutuksella, joka oli se mitä algoritmi alunperinkin käytti (ja hyvin tehokkaasti). Käytännössä sen hitaus onkin projektin suurin kompastuskivi.

Lähteet:

Python wiki

Numpy

Pygame