Haaga-Helia
University of Applied Sciences

Ohjelmistokehityksen teknologioita - *S*eminaarityö

Labyrinttipeli ja A* reitinhakualgoritmi Pythonilla ja Pygamella

6 Python

Markus Sibakov

3.5.2022

Sisältö

Tiivistelmä

Tämä työ on Pygame-pelikirjastolla tehty yksinkertaisen labyrinttipeli, joka sisältää A* reitinhakualgoritmin, jolla voidaan piirtää lyhin reitti seikkailijasta maaliin.

Tavoitteena oli edistää Python-koodaustaitoja käyttäen entuudestaan tuntematonta pygame pelikirjastoa, koska Python on vielä kielenä minulle melko uusi. Koin labyrinttipelin reitinhakualgoritmilla mielekkääksi aiheeksi, koska itse peli ei ole monimutkainen, mutta algoritmin toteutuksessa pääsee tekemään monia sellaisia asioita Pythonilla, mitä jo muilla kielillä koen hallitsevani.

Työvaiheina oli käsin tehdyn labyrintin ruudulle piirtäminen, kontrollin antaminen pelaajalle, liikkumisen rajaaminen, reitinhakualgoritmin toteutus ja lopuksi satunnaisgeneroidun labyrintin luominen.

Koska en halunnut ylittää 20 tunnin työn tekemiseen tarkoitettua aikasuositusta, en toteuttanut peliin animointeja tai ääniefektejä, vaan peli on hiljainen ja askeleet tapahtuvat välittömästi näppäimiä painaessa.

1 Johdanto/Ylätason esittely

Oma kokemukseni Python-ohjelmointikielen kanssa on pääosin saatu tämän seminaarityön sisältävällä kurssilla Ohjelmistokehityksen teknologioita. Seminaarityön aiheen valinta ei ollut helppoa, mutta halusin lisää kokemusta Pythonista ja aihe-ehdotusten myötä pygame pelikirjasto vaikutti sellaiselta työvälineeltä, josta saisin mielenkiintoisen projektin. Kuitenkin sisällön puolesta halusin, että projekti hyödyntää jotain algoritmia, joten päädyin labyrintinratkontaan.

Aihetta kesiessä suunnitelmana oli toteuttaa useampi eri algoritmi ja vertailla niitä keskenään pelin sisällä, mutta nopeasti huomasin, että onnistuakseni tekemään projektin loppuun ehdotetussa 20:ssä työtunnissa oli parempi keskittyä yhteen algoritmiin ja käyttää sitä pelaajan apuvälineenä. Aikaa kului kuitenkin pelin suunnitteluun, Pythonin ja Pygamen dokumentoinnin lukemiseen, Helsingin yliopiston kurssimateriaalin seuraamiseen, algoritmien tutkimiseen ja itse pelin ja valitun A* algoritmin toteuttamisen ja debuggaamiseen. Tärkeintä oli saada aikaan ns. valmis peli, joka hyödyntää algoritmia ja sellaisen koin saaneeni valmiiksi ehdot täyttävänä minimum viable productina, jossa toki on paljon lisättävää grafiikoiden, animaatoiden ja äänien puolesta. Yli jääneen ajan käytin

kentän satunnaisgenerointiin ja pelin uudelleenkäynnistämiseen joko generoidulla kartalla tai itse piirretyllä labyrintilla, kun nappeja painetaan.

Työvaiheet järjestyksessä olivat:

- Yliopiston kurssimateriaalin seuraaminen
- Labyrintin tarvittavan datan jäsentäminen taulukossa
- Grafiikoiden piirtäminen peliin
- Tapahtumien käsittely pelissä
- Pelihahmon liikkeiden estäminen seinien läpi
- Algoritmien tutkiminen ja valinta
- A* algoritmin pseudokoodin tutkiminen ja oman toteutuksen luominen siitä
- Algoritmin tulosten piirtäminen peliin
- Algoritmitoteutuksen testaus ja debuggaus
- Voiton julistaminen, kun pelaaja on maalissa
- Kentän satunnainen generointi
- Kenttägeneroinnin debuggaus

2 Käytetyt tekniikat

2.1 Python ja Pygame kirjasto

Pygame kirjastolla toteutettu peli oli yksi valmiista aihe-ehdotuksista, joten Pythonin käyttäminen ohjelmointikielenä oli vain luonnollista.

Pygame hoitaa peliruudun luomisen, kuvien piirtämisen, äänten soittamisen ja tapahtumien käsittelyn. Pelisilmukka kirjoitetaan siten, että peli käsittelee tapahtumat, kuten näppäinten painallukset, antaa pelin liikkuvien osien liikkua askeleen, piirtää grafiikat ja lopuksi edistää aikaa.

Toteutuksessani ei ole itsestään liikkuvia osia, joten oma silmukka ei tuota vaihetta sisällä, vaan pelin tapahtumat riippuvat kokonaan käyttäjän syötteistä.

2.2 A* reitinhakualgoritmi

A* reitinhakualgoritimi on Dijkstran algoritmiin pohjautuva lyhimmän reitin hakeva algoritmi, joka etsii solmujen verkosta lyhimmän reitin alkusolmusta maalisolmuun. A* eroaa Dijkstran algoritmista siten, että se käyttää solmujen välisten etäisyyksien lisäksi heuristista arviota solmun arvosta tarkasteltavan solmun priorisoinnissa.

A* algoritmi alkaen aloitussolumusta lisää käsittelyjonoon käsittelyssä olevan solmun kaikki yhteydessä olevat naapurisolmut. Pelin labyrintin yhteydessä naapurisolmut ovat viereiset ruudut, joilla ei ole seinää välissä käsiteltävänä olevasta ruudusta. Algoritmi merkitsee

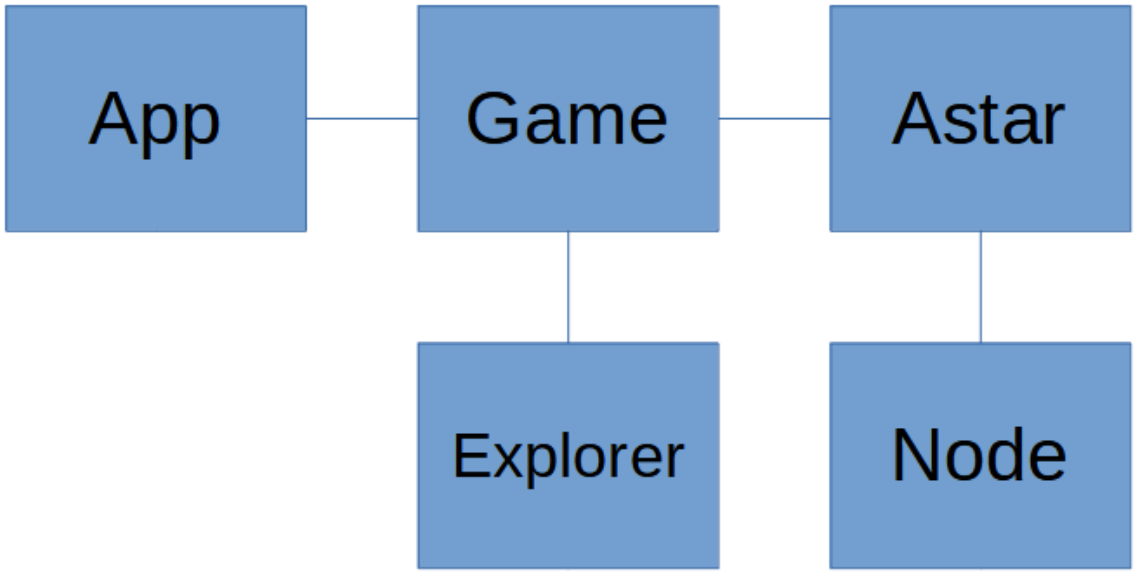
jokaiselle naapurisolmulle kuljetun etäisyyden, käsiteltävänä olevan solmun naapurisolmun vanhemmaksi, sekä heuristisen arvion, jonka laskemalla yhteen kuljetun arvon kanssa voidaan järjestää käsittelyjono arvioituun tärkeysjärjestykseen. Pelin labyrintissa jokaisen ruudun etäisyys viereiseen vapaaseen ruutuun on aina sama arvo, mutta algoritmia voidaan myös käyttää esim. kaupunkien väliseen reitinhakuun, missä kaupungit ovat maanteillä toisiinsa yhteydessä olevia solmuja.

Kun solmu on käsitelty, se merkitään käsitellyksi ja jos myöhemmin sama solmu ilmestyy toisen käsiteltävän solmun naapurina ja ilmenee, että etäisyys käsiteltävänä olevan solmun kautta on pienempi, kuin aiemmin on todettu, etäisyys päivitetään ja vanhemmuus päivitetään osoittamaan uutta käsittelyssä olevaa solmua.

Heuristinen arvio tehdään laskemalla seinistä välittämättä etäisyys alkusolmusta loppusolmuun. Koska labyrintissa voi liikkua vain neljään pääilmansuuntaan, valitsin heuristiikaksi Manhattan etäisyyden, missä lasketaan yhteen x-koordinaattien ja y-koordinaattien välinen etäisyys.

Algoritmilla on kaksi mahdollista lopputulosta. Jos maaliruutuun pääsee kävelemään aloitusruudusta, käy algoritmi läpi viimeisen solmun vanhemmuushierarkian läpi ja palauttaa siitä listan. Jos käsittelyjono tyhjenee, on se algoritmille merkki siitä, että maaliruutu ei ole saavutettavissa lähtöruudusta. (EnjoyAlgorithms)

3 Arkkitehtuurikaavio



Kuva 1 Labyrinttipelin arkkitehtuurikaavio

4 Algoritmin pseudokoodi

```
function AStar(source, destination)

  queue = set containing source
  visited = empty set

  source.g = 0
  source.f = source.g + heuristic(source, destination)

  while queue is not empty
    currentNode = pop queue element with lowest f value

    if currentNode = destination
      return construct_path(destination) // path found

    remove currentNode from queue
    add currentNode to visited

    for each neighbor in neighbors(currentNode)
      if neighbor not in visited
        neighbor.f = neighbor.g + heuristic(neighbor, destination)
        if neighbor is not in queue
          add neighbor to queue
        else
          //update g value if node already in queue
          existingNeighbour = neighbor in queue
          if neighbor.g < existingNeighbour.g
            existingNeighbour.g = neighbor.g
            existingNeighbour.parent = neighbor.parent

  return false // no path exists

function neighbors(node)
  neighbors = set of valid neighbors to node // check for obstacles here
  for each neighbor in neighbors
    if neighbor is diagonal
      //if diagonal nodes can be visited from a node
      neighbor.g = node.g + diagonal_cost
    else
      neighbor.g = node.g + normal_cost
    neighbor.parent = node
  return neighbors

function construct_path(node)
  path = set containing node
  while node.parent exists
    node = node.parent
    add node to path
  return path
```

Kuva 2 A* algoritmin pseudokoodi (EnjoyAlgorithms)

5 Reflektio

Python kielessä halusin hyödyntää luokkia, vaikkei lopullinen toteutukseni niitä varsinaisesti tarvinnutkaan. Luokkien hyödyntäminen on järkevää, jos peliä laajennettaisiin esim. lisäämällä vihollisia tai kerättäviä esineita, tai jos pelille lisättäisiin aloitusruutu, josta pelin aloituspainiketta painamalla luodaan uusi peliluokan olio.

Uutena opin Pythonista sen, että jokaisessa luokan metodissa on annettava parametri, joka kuvastaa luokan omaa tilaa, vaikka sitä ei käytettäisi. Lisäksi luokkaa luodessa sille ei voi antaa parametrina suoraan jonkun funktion palauttamaa oliota, vaan se pitää ensin tallentaa omaan muuttujaan ja vasta sen voi antaa luokan parametrina.

Algoritmia toteuttaessa aioin aluksi hyödyntää python-luokkien vapautta lisätä niihin milloin tahansa uusia arvoja, mutta se johti hankaliin ehtolausekkeisiin, joihin en ollut valmis. Käytin lopulta luokkia kuin Javassa, eli loin niille kaikki muuttujat, joita ohjelmassa tulen tarvitsemaan, mikä helpotti omaa koodin käsittelyä.

A* algoritmi näyttää pelissä käyvän ruuduissa, jotka eivät näytä loogisilta. Se johtuu yksinkertaisesti siitä, että priorisointi ei järjestä saman painoarvon solmuja millään tavalla, jolloin aiemmin yhdelle painolle merkitty node voidaan käydä läpi vaikka myöhemmin löytyy solmu, joka selvästi on heuristiikan puitteessa parempi valinta. Tämän olisi voinut parantaa järjestämällä samanpainoiset solmut käsittelyjonossa vielä niiden pelkän heuristisen arvion avulla, jolloin hassun näköisten syrjähyppyjen määrän pitäisi vähentyä entisestään.

Loppujen lopuksi en hyödyntänyt kovin montaa aspektia Pygame pelikirjastosta. Vain tärkeimmät funktiot tuli käytettyä pelissä, eli ruudun luominen, näppäinten käsittely ja piirtäminen. Merkittävä osa ajasta kului algoritmin toteuttamisessa.

Jos käyttäisin sulavaa liikkumisanimointia tai jotain itsenäisesti tapahtuvaa prosessia, voisi pelisilmukassa olla tapahtumakäsittelyn ja piirtämisten välissä funktio, joka suorittaa yhden askeleen näistä toimenpiteistä. Sulavan liikkumisen olisi voinut toteuttaa niin, että tapahtumankäsittelyssä asetetaan kohdekoordinaatit ja tämä funktio sitten siirtäisi seikkailijaa joka kierroksella muutaman pikselin kohdekoordinaattia kohti. Aika ei kuitenkaan riittänyt tämän lisäämiseen peliin.

Peliä voisi laajentaa tekemällä labyrintin generaatioalgoritmin, joka osaa tehdä varmuudella ratkaistavissa olevia labyrintteja haluttujen parametrien mukaisesti. Parametrina voisi olla mm. onko labyrintissa useita reittejä maaliin, jotta lyhimmän reitin hakeva algoritmi voi ehdottaa erilaisia reittejä eri aloitusruuduissa. Peliä voisi myös kehittää lisäämällä sinne

mielekästä interaktiota kerättävien tavaroiden, ansojen, vihollisten, tönittävien laatikoiden, yksisuuntaisten seinien yms. avulla.

Lähdeluettelo

EnjoyAlgorithms. *A* search algorithm.* Noudettu osoitteesta https://www.enjoyalgorithms.com/blog/a-star-search-algorithm