Verneri Turunen, 649 403
ISM, kandidaatin tutkinto
Aalto yliopiston kauppakorkeakoulu
6.5.2020

Y2 Projekti: Labyrintti

Yleiskuvaus

Ohjelma luo ja piirtää ruudulle labyrintin, ja asettaa hiiren labyrintin keskelle vapaaseen ruutuun. Maaliruutu sijaitsee labyrintin oikeassa alakulmassa. Labyrintti on weave-tyyppinen, eli se sisältää ali- ja ylikulkuväyliä. Peli on tallennettavissa tiedostoon, ja peli voidaan ladata tiedostosta.

Poiketen alkuperäisestä suunnitelmasta, labyrintti on pyritty toteuttamaan vaikean vaikeustason vaatimuksiin. Labyrintti on toteutettu graafisella käyttöliittymällä. Käyttäjä määrittelee itse labyrintin koon ohjelman alussa, ja hiirtä ohjataan nuolinäppäimien avulla.

Labyrintissa on kiinnitetty erityishuomiota pelilliseen käytettävyyteen: avautuvalla applikaatioruudulla näkyvien labyrintin ruutujen koko skaalautuu käyttäjän valitsemien kokomuuttujien perusteella. Ruudun oikeaan yläkulmaan on toteutettu valikko, josta käyttäjä pystyy lopettamaan tai tallentamaan pelin, pyytämään vihjettä, tai turhautumisen yltyessä siirtyä godmodeen. Jokainen nappi kertoo käyttäjälle toiminnastaan, kun hiiri asetetaan niiden kohdalle. Tallentaminen sujuu helposti avautuvan ikkunan kautta, ja tallentaminen on mahdollista myös lopettamisen jälkeen. Lopettamalla pelin ohjelma vie pelaajan maaliin, näyttäen lyhyimmän reitin pelaajan sen hetkisen sijainnin perusteella, merkaten ensimmäinen askeleen punaisella. Labyrintin kokoa ei ole rajoitettu vain näkyvään alueeseen, vaan se liikkuu ruudulla pelaajan mukana.

Ratkaisu ja luontialgoritmien toteutus on pilkottu erillisiin metodeihin, joten niiden lisääminen ja modifiointi onnistuu helposti. Maze-luokka luo ensin kokovaatimusten mukaisen labyrintin Square-olioista, joilla on kaikki seinät. Tämä mahdollistaa muiden passage carver – algoritmien helpon implementoinnin.

Käyttöohje

Ajaakseen ohjelman käyttäjällä tulee olla ladattuna tietokoneelleen python 3 sekä PyQt5. Ladattuaan ohjelman tiedostot (main.py, gui.py, corrupted_maze_file_error.py, maze.py, square.py), käyttäjä voi käynnistää ohjelman joko IDE:ssä, tai terminaalin kautta, komennolla python3 filepath/main.py, jossa filepath nimensä mukaisesti on ohjelmakansion sijainti tietokoneella.

Käynnistyttyään ohjelma tiedustelee ensin, haluaako käyttäjä aloittaa uuden pelin, vai ladata pelin tiedostosta. Aloitettaessa uusi peli, ohjelma kysyy käyttäjältä käyttäjänimen (syötettäessä tyhjä käyttäjä ohjelma antaa käyttäjän hiirelle vakiosymbolin). Tämän jälkeen käyttäjä valitsee labyrintille leveyden ja pituuden. Ladatessa peli tiedostosta, ohjelma kysyy tiedoston nimeä, ja syötettäessä oikealla formaatilla rakennetun tallennustiedoston tiedostosijainti, peli alkaa.

Käyttäjälle aukeaa applikaatioikkuna, jonka kautta pelaaminen tapahtuu nuolinäppäimien avulla. Normaalit viivat ovat seiniä, joiden läpi ei voi kulkea. Katkoviivat kuvaavat alla kulkevaa tunnelia. Tavoitteena on päästä oikeassa alakulmassa sijaitsevaan maaliruutuun. Painamalla oikean yläkulman valikosta quit-nappia, ohjelma näyttää käyttäjälle lyhyimmän reitin maaliin, ja peli päättyy. Kyseisen labyrintin voi edelleen ladata lopetuksen jälkeen save-napista, jolloin ladattaessa peli kyseisestä tiedostosta hiiri asetetaan samaan kohtaan, missä se sijaitsi ennen lopetusta. Kuten jo mainittu, tallennus tapahtuu helposti kirjoittamalla tiedostonimi save-nappia painamalla aukeavaan ikkunaan.

Ohjelma tarjoaa käyttäjälle mahdollisuuden pyytää vinkkiä vaikean labyrintin ratkaisemiseen klikkaamalla ask for tip -nappia: aukeava ikkuna antaa tällöin käyttäjälle noin 20% jäljellä olevista askeleista lyhyimmällä reitillä. Jos tämäkään ei riitä, käyttäjä voi klikata god-mode

nappia, jolloin seinätkään eivät voi enää pysäyttää hiiren etenemistä. God-modesta voi poistua klikkaamalla samaa painiketta uudestaan. Maaliruutuun tulee saapua normaalitilassa, jotta peli rekisteröi saapumisesi maaliin.

Ulkoiset kirjastot

Ohjelman toteutuksessa käytettiin Pythonin standardikirjaston lisäksi PyQt-käyttöliittymäkirjastoa graafisen käyttöliittymän toteutukseen, random-kirjastoa labyrintin luontialgoritmiin, sekä queue-kirjastoa labyrintin ratkaisualgoritmiin.

Ohjelman rakenne

Ohjelma rakentuu luokista Maze, Square, GUI, CorruptedMazeFileError, Test sekä main-luokasta. Seuraavassa tiivistelmä kunkin luokan roolista ohjelman toiminnassa:

Maze: ohjelman tärkein luokka, joka huolehtii projektin ydinongelman ratkaisusta – labyrintin luomisesta. Uusi Maze-olio saa parametreinaan labyrintin mitat, sekä hiiren symbolin, joiden perusteella luodaan kaksiulotteinen lista Square-luokan olioista. Tärkeimmät metodit ovat labyrintin luontiin käytettävä **carve_maze**, sekä lyhyimmän reitin maaliruutuun löytävä **find_shortest_path**. Kumpikin näistä metodeista käyttää useita apumetodeja, joista lisää algoritmeja käsittelevässä kappaleessa. Muita hyödyllisiä metodeja luokassa on mm. **square_in_bounderies**, joka tarkistaa, onko ruutu pelin rajojen sisäpuolella, **get_mouse_square,** joka palauttaa ruudun, jossa pelaaja sijaitsee, sekä tiedoston käsittelyyn tarkoitetut **save_to_file** ja luokkametodi **load_from_file**. Näin jälkiviisaana, nämä olisi ehkä ollut järkevämpää toteuttaa omaan luokkaansa.

Square: Square-oliot muodostavat koko labyrintin rakennuspalikat. Labyrintin koostuminen neliöistä mahdollistaa helpon seinien ja hiiren sijainnin manipulaation eri metodeilla. Tärkeimpiä metodeja luokassa ovat labyrintin luonnissa hyödynnettävät **remove_wall** (poistaa seinän kahden neliön välistä) ja **has_not_been_visited** (tarkistaa, onko tietty neliö jo lisätty luontialgoritmin hyödyntämään listaan). Weave-labyrintin luontia avustavat

vertical_passage ja **horizontal_passage**, joiden avulla tarkistamme, onko alikulku mahdollinen. **Add_under_square** lisää nimensä mukaisesti neliölle toisen neliön samoilla koordinaateilla edustamaan alikulkua.

GUI: Graphical User Interface -luokka luo PyQt5:n avulla applikaatioikkunan ja piirtää luodun Maze-olion mukaisen labyrintin, jonka välityksellä käyttäjä ratkoo labyrinttia. Tärkeimpiä metodeja ovat **paintEventin** hyödyntämät **drawWalls** ja **drawSolution**, joiden avulla ikkunan grafiikka seuraa hiirtä, tehden isojenkin labyrinttien pelaamisesta mahdollisen, sekä näyttää ratkaisussa koko labyrintin samassa ikkunassa. Käyttäjän syöte nuolinäppäimillä luetaan **keyPressEventin** avulla. Jokaiselle valikon toiminnolle on toteutettu omat metodinsa. Hiiri esitetään **draw_mousen** avulla, joka saa parametreinaan sekä ruudulle sijoittamisen kannalta oikeat x- ja y-koordinaatit, sekä hiirineliön koordinaatit itse labyrintilta. Metodi esittää toisen neliön alla olevan hiiren pienempänä.

CorruptedMazeFileError: Luokka on luotu auttamaan Maze-luokan load_from_file-metodissa auttamaan virheiden käsittelyssä. Virheellisen tiedoston kohdalla luokka tulostaa ruudulle virheilmoituksen, joka auttaa käyttäjää tunnistamaan virheen alkuperän.

Test: Testiluokka Maze- ja Square-luokkien tärkeimpien metodien yksikkötestaukseen.

Main: Käynnistää ohjelman, vastaanottaa (ja käsittelee virheelliset) käyttäjän syötteet, luo labyrintin ja käynnistää applikaation.

Algoritmit

Miettiessäni labyrintin luontialgoritmia projektin alussa, päädyin valitsemaan jonkin ”passage-carver” metodeista, sillä tämä mahdollisti labyrintin luomisen aluksi kokovaatimusten mukaiseksi squareista, joilla on vielä alkuvaiheessa kaikki seinät. Näin sain myös eriytettyä carve_maze metodin, mikä mahdollistaa muiden algoritmien helpon implementoinnin. Päädyin valitsemaan growing tree – algoritmin, koska se näytti olevan helposti laajennettavissa luomaan haluamani weave-labyrintti alikulkuineen. Muita vaihtoehtoja olisi ollut esimerkiksi recursive backtracker sekä growing forest (growing tree laajennus).

Algoritmi arpoo ensin random-kirjaston avulla satunnaisen neliön, josta reitin luominen starttaa. Tämä neliö lisätään listaan. Algoritmi hyödyntää **unvisited_neighbours** -metodia, ja arpoo uuden neliön lisättäväksi listaan, jolloin näiden neliöiden välinen seinä poistetaan, ja tämä lisätty neliö siirtyy tutkittavaksi. Mikäli naapureita ei senhetkisellä neliöllä ole, katsotaan, onko alikulku mahdollinen. **Able_to_carve_under** tarkistaa, onko naapurineliö rajojen sisällä, onko naapurineliöllä käytävä kohtisuorassa menosuuntaamme verrattuna (näin varmistamme, että labyrintista tulee visuaalisesti selkeä, eikä neliöstä ole mahdollista ”pudota toiseen neliöön”), sekä sen, onko mahdollisen alikulun toisella puolella oleva neliö samassa suunnassa vielä vierailematta. Mikäli nämä ehdot toteutuvat, algoritmi lisää viereiselle neliölle under_squaren, ja jatkaa vierailemattomista naapureista alikulun toisella puolella. Kun naapureita ei ole ja alikulkua ei onnistu, lista poistaa neliöitä siihen asti, kunnes naapureita löytyy, tai lista on tyhjä, jolloin labyrintti on valmis.

Ratkaisualgoritmiksi valitsin Breadth first search -algoritmin. Halusin ratkaisualgoritmin, joka löytää aina lyhyimmän ratkaisun. Ryhdyttyäni jo töihin, ymmärsin että algoritmi on itsessään hyvin tehoton, koska se lisää jonoon edelleen reittejä, jotka ovat kohdanneet umpikujan. Tämän vuoksi tein siihen pienen muutoksen, jolla jono ei lisää takaisin reittejä, jotka yrittävät kääntyä takaisin edellisen askeleen suuntaan (lyhyin reitti ei koskaan sisällä tällaista koukkausta). Algoritmi lisää jonoihin koordinaateista muodostuvia stringejä, joiden perusteella muodostetaan valideja polkuja hiiren senhetkisestä sijainnista, kunnes kohdataan maaliruutu. Metodi **validMove** lisää kuhunkin merkkijonoon mahdolliset kulkusuunnat, ja myös huolehtii siitä, että toimimme oikein under_squarien kanssa. **FindGoal** tarkistaa, onko reitti löytänyt maaliruudun, ja **find_shortest_path** palauttaa näitä metodeja hyödyntäen lyhimmän reitin. Vaihtoehtoja ratkaisualgoritmeihin oli monia, mutta olin kiinnostunut queuen käytöstä, joten päädyin valitsemaan Breadth first searchin.

Tietorakenteet

Tärkeimmät ohjelmassa käytetyt tietorakenteet ovat list ja dictionary, joista kumpikin ovat mutable-tietotyyppejä. Niiden avulla toteutettiin labyrintin luontialgoritmi. Ratkaisualgoritmissa hyödynnettiin queue:n avulla stack-tietotyyppiä sekä jonoon lisättäviä stringejä. String on sen sijaan immutable tietotyyppi. Näistä tärkeimpänä on ehdottomasti nostettava esiin

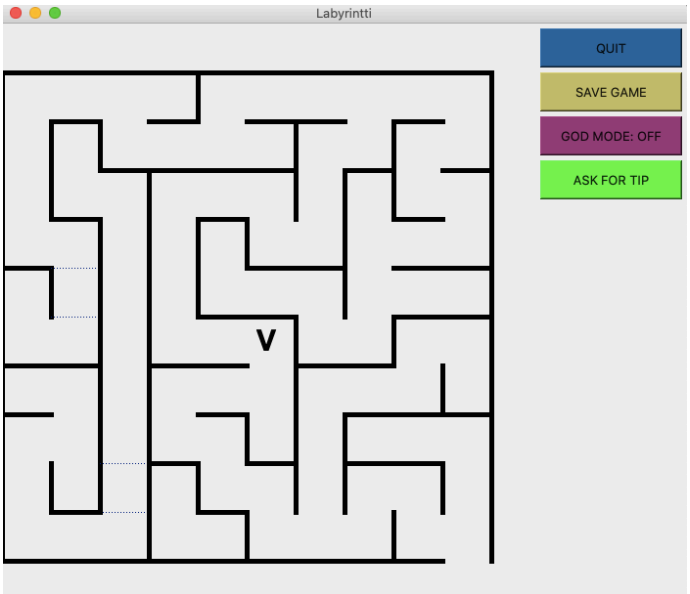
listojen käyttö, dictionaryt, stringit olisi ollut mahdollista korvata myös muilla ratkaisuille, ja stack olisi voitu toteuttaa ratkaisualgoritmissa myös esimerkiksi listoilla.

Tiedostot

Ohjelma käsittelee tekstitiedostoja, johon peli on mahdollista tallentaa. Peli tallennetaan seuraavassa tiedostomuodossa: ensimmäiselle riville labyrintin, leveys, pituus sekä hiiren symboli, seuraaville riveille squaret muodossa 1011F/, jossa 1 merkkaa seinää ja 0 seinän puuttumista, F/T merkkaa sitä, onko squarella under_square, ja ’/’ erottaa squaret toisistaan (myös rivi päättyy ’/’-symboliin). Viimeisellä rivillä merkataan hiiren sijainti squaressa tai under_squaressa (S = surface, U = under), sekä hiiren x- ja y-koordinaatit muodossa S/1/1/.

Esimerkkinä, seuraavan lainen tekstitiedosto luo alla olevan labyrintin:

```
10/10/V
1001F/1010F/1000F/1110F/1001F/1010F/1010F/1000F/1010F/1100F/
0101F/1101F/0011F/1010F/0010F/1110F/1001F/0110F/1001F/0110F/
0101F/0011F/1100F/1001F/1010F/1100F/0101F/1101F/0011F/1100F/
0011F/1100F/0101F/0101F/1101F/0011F/0110F/0001F/1010F/0110F/
1101F/0101T/0101F/0101F/0011F/1010F/1100F/0001F/1010F/1110F/
0011F/0110F/0101F/0011F/1010F/1100F/0011F/0110F/1001F/1100F/
1011F/1100F/0101F/1001F/1010F/0100F/1001F/1010F/0110F/0111F/
1001F/0100F/0101F/0011F/1100F/0111F/0101F/1011F/1010F/1100F/
0101F/0111F/0101T/1101F/0011F/1100F/0101F/1001F/1100F/0101F/
0011F/1010F/0110F/0011F/1110F/0011F/0010F/0110F/0011F/0100F/
S/5/5/
```



Testaus

Testaus toteutettiin melko lailla suunnitelman mukaisesti, eli uusien metodien ja luokkien toimintaa testattiin jatkuvasti läpi projektin. Tärkein testauksen muoto oli jatkuva ohjelman ajaminen ja korjausten tekeminen kokeilun ja epäonnistumisen kautta. Loppupuolella toteutettiin myös yksikkötestausluokka, jonka avulla varmistettiin tärkeimpien Maze- ja Square-luokkien toimivuutta. Lopputulos poikkesi melko paljon suunnitellusta helppotasoisesta toteutuksesta, minkä takia erityisesti GUI-luokan testaukseen ja toteutukseen kului aikaa.

Ohjelman tunnetut puutteet ja viat

Vikoina mainittakoon graafisessa toteutuksessa ruutujen katoaminen oikealta, kun käyttäjä etenee vasempaan reunaan. Tämä ei varsinaisesti haittaa pelaamisesta, mutta vähentää jonkin verran ohjelman estetiikkaa. Toisena puutteena on tiedoston käsittelyssä se, ettei tämän hetkinen tiedoston latausmetodi osaa tarkistaa, onko tekstitiedostona syötettävässä labyrintissa ”järkeä”, käyttäjä voi esim. syöttää tiedoston, jossa kaikilla squareilla on under_square, tai niin että undersquaret eivät noudata carve_underin sääntöjä. Metodi osaa kuitenkin tarkistaa suurimman osan yleisistä vioista, kuten perinteiset kirjoitusvirheet.

3 parasta ja 3 heikointa kohtaa

Parhaana ominaisuutena ohjelmassani näen ehdottomasti helpon pelattavuuden, ja toteuttamani pelivalikon. Oma henkilökohtainen suosikkini on god-mode, jonka toteuttaminen oli mielessäni jo jonkin aikaa. Hyvänä ominaisuutena näen myös alikulkujen toteutuksen katkoviivoilla sekä pienempänä hiirisymbolina. Kolmantena mainittakoon valikon hover-toiminto sekä hiirisymbolin muuttuminen god-modeen siirryttäessä.

Heikkouksina mainittakoon jo puutteissa todetut grafiikan katoaminen ruudun oikeasta laidasta vasemmalle mentäessä, tiedostosta lataamisen pienoiset puutteet, sekä Maze-luokan

paisuminen ehkä vähän liian massiiviseksi; erityisesti tiedoston käsittelyn olisin paremmalla suunnittelulla siirtänyt omaan luokkaansa.

Poikkeamat suunnitelmasta

Kuten jo mainittu, suunnitteluvaiheessa lähtökohtana oli helppotasoisen projektin toteuttaminen. Koodausinto kuitenkin kasvoi tehdessä, ja ajan riittäessä päätös kallistuikin haastavampaan. Vielä suunnitteluvaiheessa algoritmit olivat hämärän peitossa, joten näin jälkikäteen katsottuna, en varsinaisesti poikennut valtavasti suunnitelmassa mainitusta työjärjestyksestä, mutta suurin työ ja aika kului loppujenlopuksi ominaisuuksiin, joista en vielä ollut tietoinen tuolloin (erityisesti PyQt:n opetteleminen). Aikaakin kului luonnollisesti huomattavasti suunniteltua enemmän, päädyttyäni toteuttamaan haastavaa projektia.

Toteutunut työjärjestys ja aikataulu

Työskentelin projektin parissa tiiviissä ”ryppäissä”. Ensimmäinen rypäs sijoittui noin viikkoa ennen check-up päivämäärääni 31.3, jolloin työskentelin tiiviisti Maze- ja Square-luokkien parissa, ja sain luotua vahvan pohjan loppuprojektilleni. Check-upin jälkeen koodausintoa riitti, ja etenin muutaman päivän täysinäisellä panostuksella graafiseen toteutukseen. Ratkaisualgoritmin toteutus tapahtui vasta siinä vaiheessa, kun grafiikka toimi jo kohtalaisen hyvin.

Toinen rypäs alkoi noin viikkoa ennen palautuspäivämäärää, jolloin hioin main-luokkaa, toteutin tiedostoon tallennuksen, panostin haastavan tason vaatimuksiin pelattavuuden ja liikkuvan grafiikan suhteen , toteutin testiluokan sekä tarkistelin ja siistin koodia. Kaiken kaikkiaan aikaa kului melko paljon suunniteltua kauemmin, mutta koin haastavamman työn toteutuksen palkitsevana.

Arvio lopputuloksesta

Yhteenvetona voin todeta, että projekti oli todella opettavainen, ja olen tyytyväinen siihen, että päädyin sittenkin toteuttamaan vaikeampaa labyrinttia. En päätynyt ratkaisuun niinkään paremman arvosanan kiilto silmissä, vaan oppimisen ja onnistumisten tuoman hyvän fiiliksen takia. Olen kaiken kaikkiaan hyvin tyytyväinen lopputulokseen. Oli hauskaa koodata valikkoa ja erillisiä pikku yksityiskohtia (hover-toiminto, hiiren symbolin muutos god-modessa, valikon uudelleen järjestely quitin yhteydessä, pop-up viestit) ja nähdä lopputulos omalla ruudulla. Projekti oli ensimmäinen askeleeni itse toteutettujen algoritmien maailmaan, ja tuntui mahtavalta, kun huomasi että metodit tekevät juuri sitä mitä pitää. Heikkouksina mainitut pienet puutteet grafiikassa ja tiedostonkäsittelyssä olisi ollut osaamisellani korjattavissa, mutta loppusuoralla aika (ja hieman myös jaksaminen) tulivat vastaan.

Viitteet

Kirjastot:

https://docs.python.org/3/library/index.html#the-python-standard-library

https://docs.python.org/3/library/random.html

https://docs.python.org/3/library/queue.html

Weave maze lähteet:

https://weblog.jamisbuck.org/2011/3/4/maze-generation-weave-mazes.html

https://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm#

http://www.astrolog.org/labyrnth/algrithm.htm

Ratkaisualgoritmi:

https://en.wikipedia.org/wiki/Breadth-first_search

https://techwithtim.net/tutorials/breadth-first-search/