1. Henkilötiedot

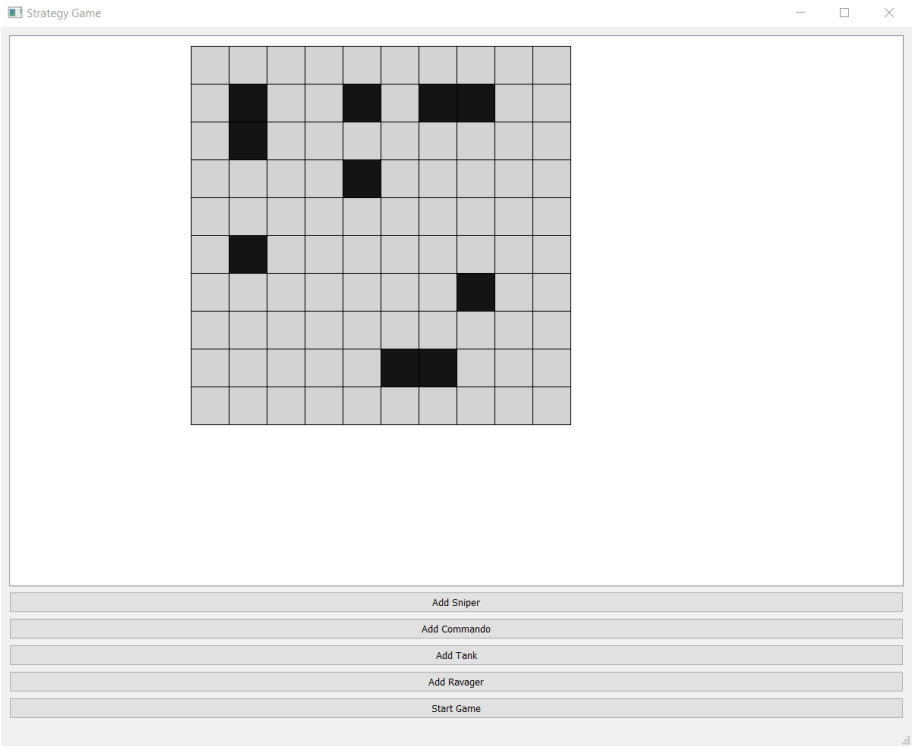
Strategiapeli | Kim Valén| Bioinformaatioteknologia | 26.4.2021

2. Yleiskuvaus

Vuoroperäinen strategiapeli. Pelikenttänä toimii kaksiulotteinen taulukko, johon voi sijoittaa omia yksiköitä pelin alussa. Yksikköjä on käytettävä toisen osapuolen tuhoamiseen. Kentälle generoidaan pelin alussa satunnaisesti esteitä, jotka on pakko kiertää. Projekti on toteutettu keskivaikealla vaikeustasolla, mutta se sisältää esimerkiksi vaikean vaikeustason kriteereistä ajanotto-ominaisuuden ja osittain hahmojen eri käyttäytymiset. Pelissä on 4 eri yksikkötyyppiä. Kentälle voi asettaa enintään 5 yksikköä.

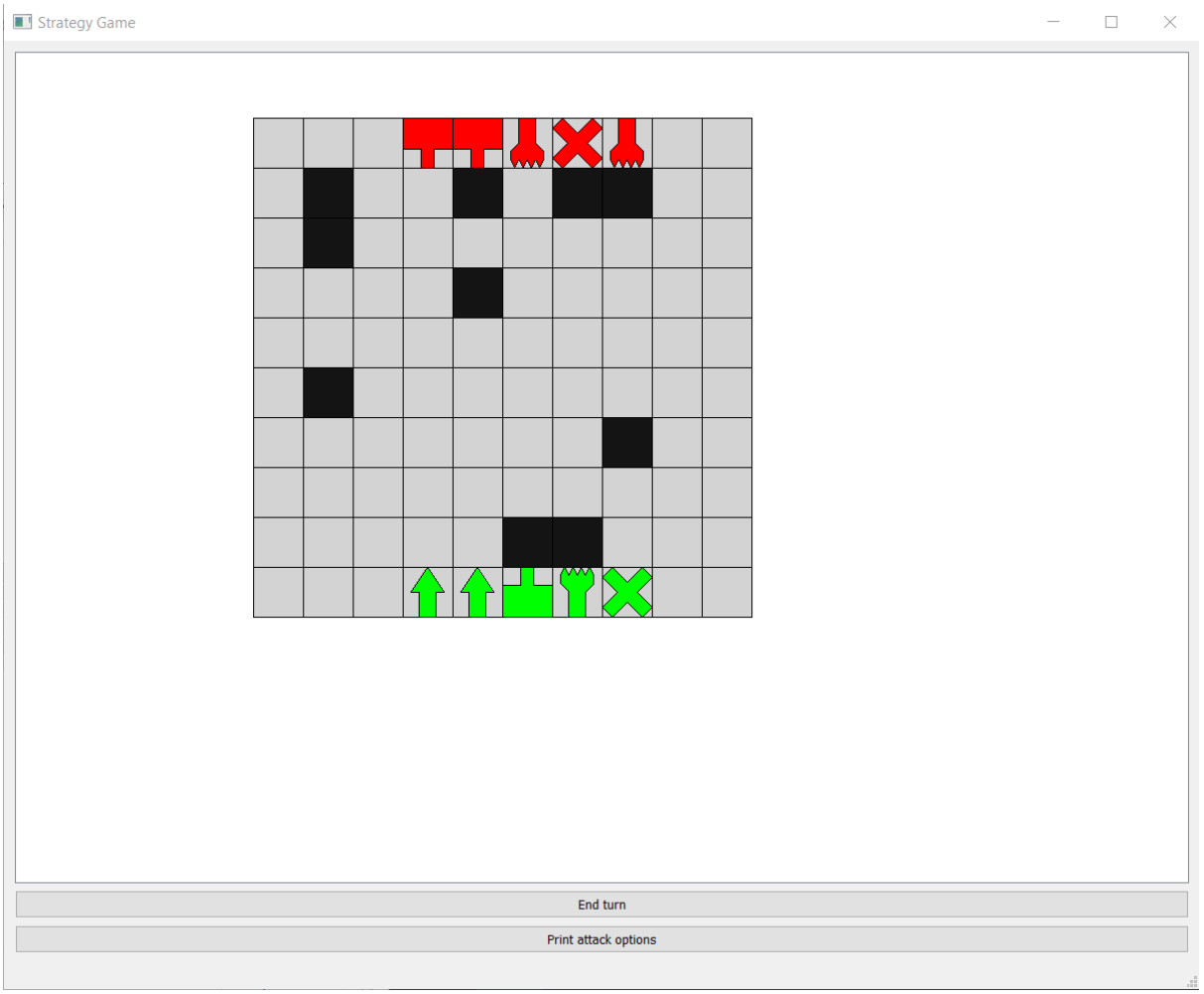
3. Käyttöohje

<u>Ajaminen</u>
Ohjelma vaatii, että käyttäjä on asentanut PyQt5 käyttöliittymän tietokoneelleen. Ohjelma käynnistetään ajamalla main.py-tiedosto. Peli on suunniteltu kentille, joiden koko on vähintään 10x10 (ohjelma ajetaan tällä oletusmittakaavalla). Halutessaan käyttäjä voi muuttaa näitä luomalla World-olion (tiedostossa main.py) eri leveys- ja korkeusarvoilla (ei suositeltavaa, sillä yksikköjen vahvuudet riippuvat kentän koosta).



Pelin aloittaminen

Jokaista yksikköä varten on oma näppäin, jota painamalla yksi yksikkö lisätään kentälle. Käyttäjä voi sijoittaa enintään 5 yksikköä kentälle ja aloittaa pelin ‘Start-game’-painikkeesta.



Näkymä painaessa ‘Start game’

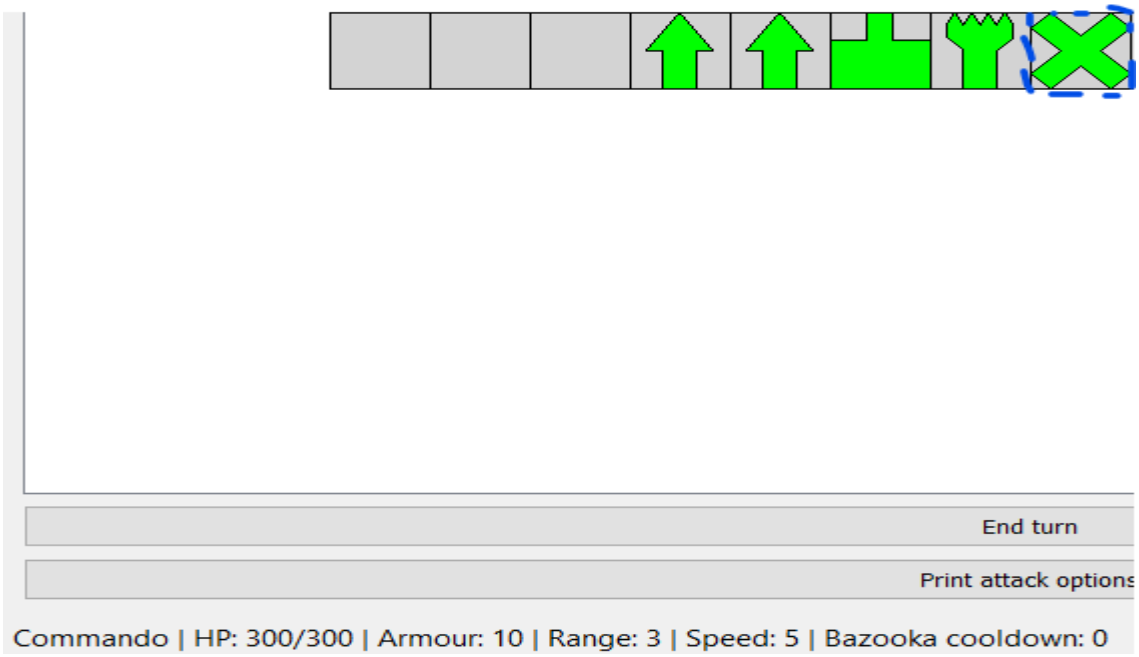
Pelin säännöt

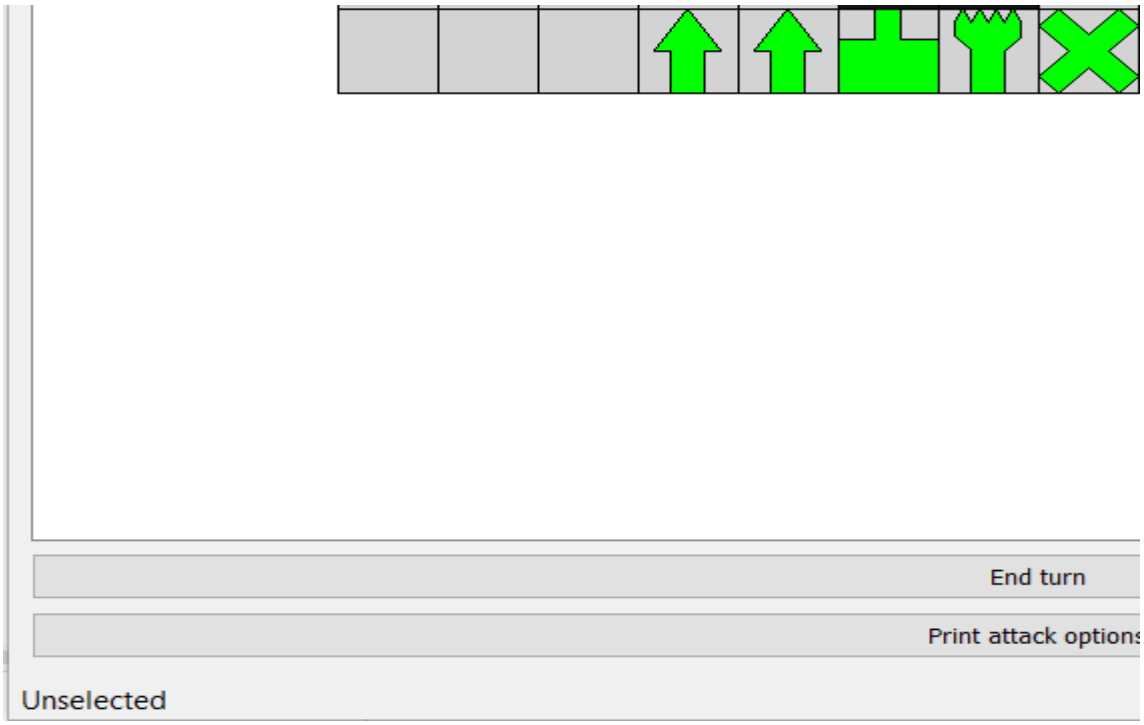
- Vuoron aikana pystyt liikuttamaan yhtä yksikköä (ei pakollista)
- Pelissä ei voi liikkua diagonaalisesti
- Vuoron aikana pystyt hyökätä yhdellä yksiköllä (ei pakollista)
- Vuoro vaihtuu painamalla ‘End turn’
- Nappi ‘Print attack options’ tulostaa kentällä olevien yksiköiden hyökkäystyypit
- Pelin voittaa tuhoamalla vastustajan joukot ja painamalla ‘End turn’

Yksikön tiedot

Painamalla kentällä olevaa yksikköä ikkunan alareunaan ilmestyy kyseistä yksikköä kuvaava teksti, joka sisältää mm. yksikön nimen, elämäpisteet, kantaman, jolle se pystyy ampumaan (range), kuinka pitkän matkan se pystyy liikkumaan (speed), panssarointipisteet ja aseiden jäähtymisajan vuoroina (cooldown).

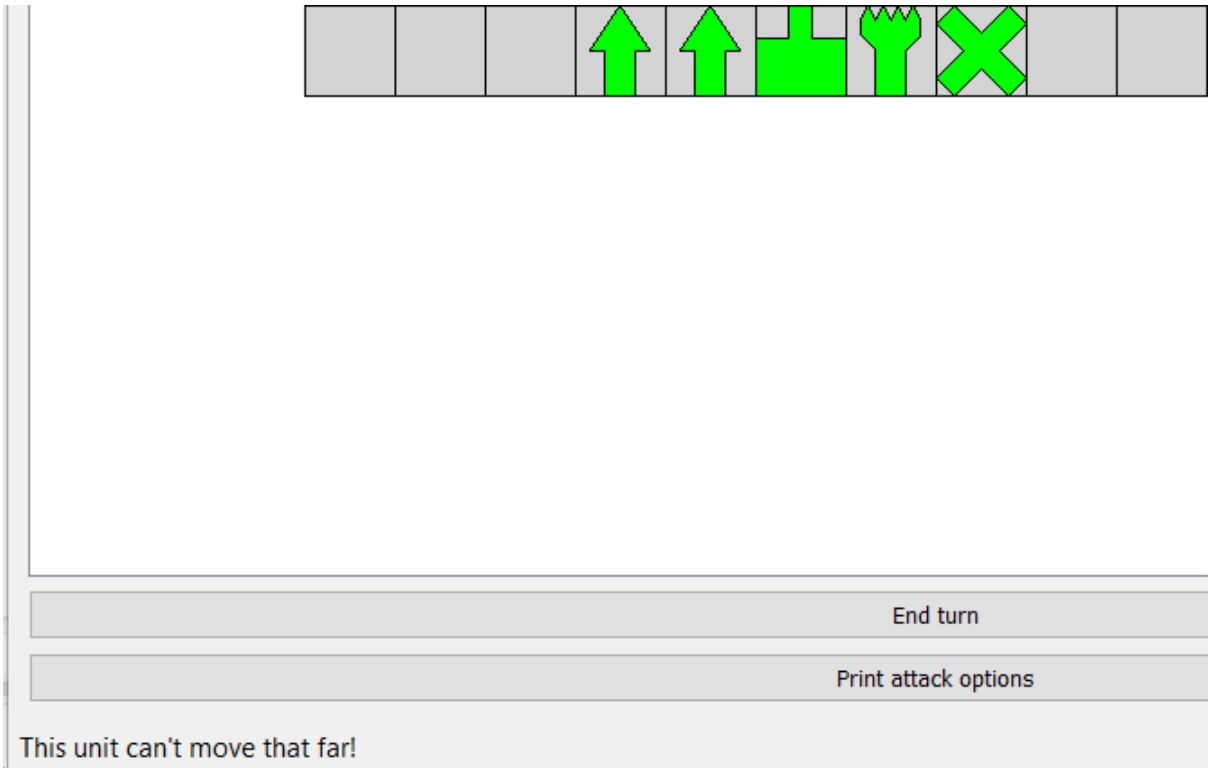
Yksikön klikkaaminen ‘valitsee’ sen - uudelleenklikkaaminen poistaa valinnan ja antaa ilmoitusviestin ‘Unselected’.





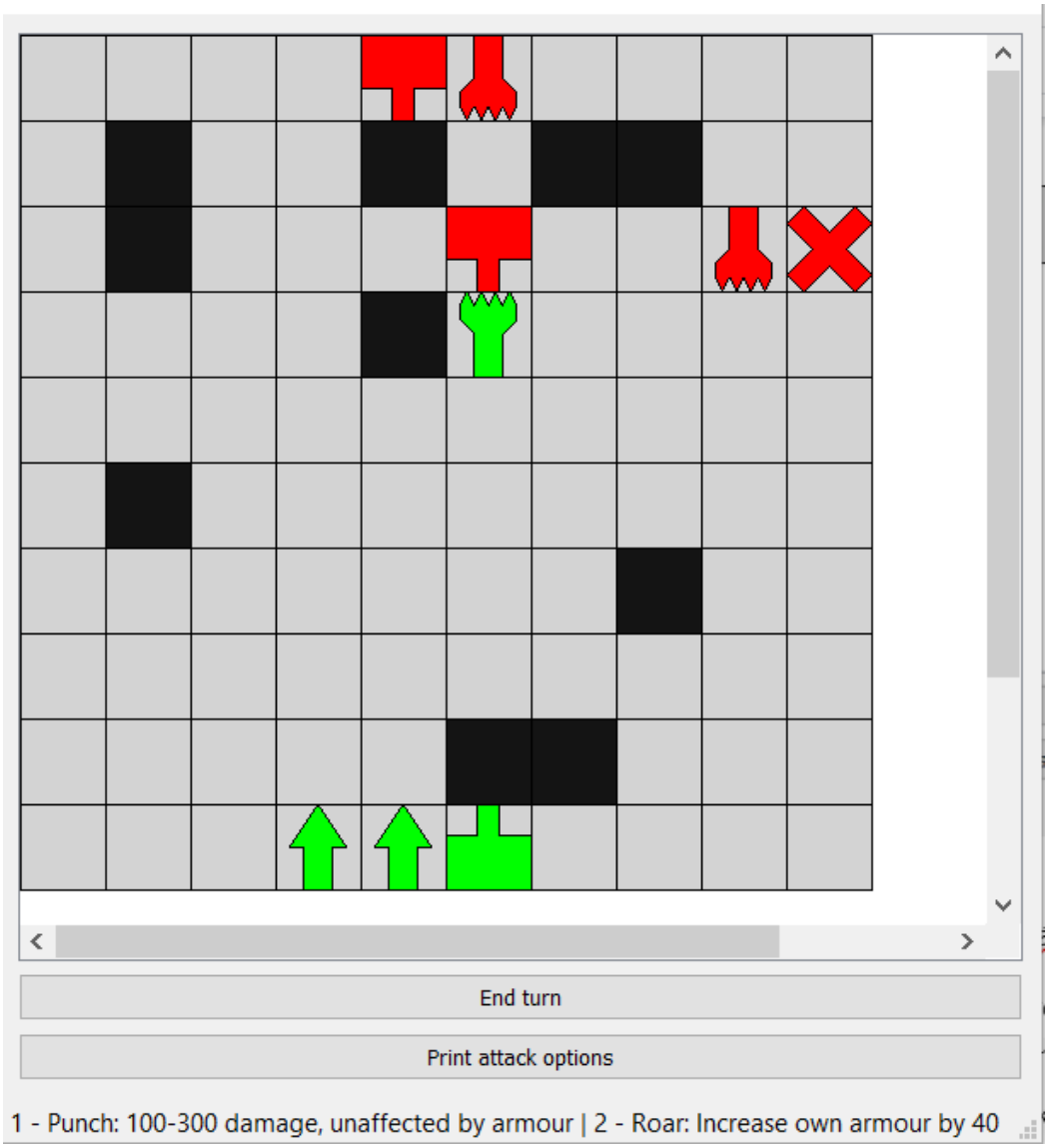
Liikkuminen

Valitse haluamasi yksikkö klikkaamalla sitä ja klikkaa sitten ruutua johon haluat sen liikkuvan. Ikkunan alareunaan tulee asianmukainen virheilmoitus, mikäli tämä ei ole mahdollista



Hyökkääminen

Valitse haluamasi yksikkö klikkaamalla sitä. Klikkaa sitten vihollisyksikköä, jota haluat hyökätä. Jos tämä ei ole mahdollista, ikkunan alareunaan tulee asianmukainen virheilmoitus. Jos hyökkäys on mahdollinen, ikkunan alareunaan ilmestyvät eri hyökkäysvaihtoehdot (joita on maksimissaan kolme) ja niiden kuvaus. Eri yksiköillä on eri hyökkäysvaihtoehtoja. Hyökkäys suoritetaan painamalla näppäimistön numeroa 1, 2 tai 3, joista kukin on kytköksissä eri hyökkäykseen. HUOM, yksiköiden kantama (range) lasketaan seuraavasti: $\min(\Delta x, \Delta y)$



4. Ulkoiset kirjastot

Käyttöliittymän luontiin käytetään kirjastoja PyQt5 ja sys. Ajanottoa varten kirjastoa timeit, jolla voi tallentaa kellonajan pelin alussa, sekä lopussa. Pelin kesto saadaan näiden aikojen erotuksesta. Esteiden generoimiseen ja yksikköjen aiheuttaman vahingon laskemiseen on käytetty kirjastoa random, sillä se mahdollistaa helpon ja nopean tavan luoda pseudosatunnaisia lukuja. Kirjastoa queue käytetään lyhyimmän reitin löytävässä algoritmissa prioriteettijonon luomiseksi. Prioriteettijono on tietorakenne, joka voidaan toteuttaa monella eri tavalla (esimerkiksi listana), mutta queue-kirjaston luokka PriorityQueue oli minulle entuudestaan tuttu, joten päätin käyttää sitä. Tekoälyn hyökkäysmetodit perustuvat hyökkäysten simulointiin (enemmän tästä kohdassa ‘6. Algoritmit’). Tätä varten ohjelmassa käytetään

kirjastoa copy ja sen metodia deepcopy, joka luo kopiot hyökkäävästä ja puolustavasta oliosta. Kirjasto copy kuuluu Pythonin standardikirjastoihin.

5. Ohjelman rakenne

Ohjelma jakautuu kahteen osakokonaisuuteen: käyttöliittymään liittyvät luokat ja taustalla tapahtuviin prosesseihin liittyvät luokat.

Käyttöliittymästä vastaavat luokat:

GUI
Ohjelman keskeisin luokka. Luo peli-ikkunan ja scenen, jossa pelin visuaalinen kuvaus tapahtuu. Luo näppäimet (buttons) ja yksikköjen graafiset itemit sekä poistaa ne niiden tuhoutuessa. Vastaa yksikköjen liikuttamisesta ja niiden kanssa vuorovaikuttamisesta. Vastaa vuoron päättymisestä, ajanotosta ja pelin päättymisestä. Kaikki luokan metodit toimivat keskeisenä osana ohjelman toimintaa. **GUI-luokalla on yhteys luokkiin Game, World ja UnitGraphicsItem.**

UnitGraphicsItem
Luo jokaiselle yksiköille omanlaatuisen näköisen itemin ja lisää sen sceneen. Pelaajan yksiköt ovat vihreitä ja vastustajan punaisia. Vastaa myös vastustajan yksiköiden kääntämisestä ympäri. Luokan tärkein ominaisuus on yksikön metodi update_position, joka päivittää yksikön visuaalisen itemin sijainnin kentällä ja metodi mousePressEvent, jonka avulla yksikön kanssa voi vuorovaikuttaa. **Luokalla on yhteys luokkiin GUI ja Unit (ja sen alaluokkiin).**

SquareGraphicsItem
Perii luokan QGraphicsItem. Jokaista ruutua varten luodaan oma olio. Esteet värjätty mustiksi, vapaat ruudut harmaiksi. Jokaisen ruutuolion sijainti kentällä saadaan mousePressEvent-metodilla. Näin klikkaamalla tiettyä ruutua voidaan liikuttaa siihen mahdollisesti valittu yksikkö. **Luokalla on yhteys luokkaan GUI.**

Taustaluokat:

World
Taustaluokista keskeisin. Vastaa kentän ja esteiden generoinnista, kentässä olevien yksiköiden seuraamisesta, niiden lisäämisestä ja poistamisesta. Luokassa World oleva metodi add_units_to_battlefield() on kirjoitettu siten, että pelin alussa yksiköt sijoittuvat kentän ensimmäisen rivin keskelle. Luokkaan on myös toteutettu metodi

line_of_sight, joka määrittää sen kykeneekö jokin yksikkö näkemään toisen yksikön tietystä sijainnista (erittäin keskeinen pelin kulun kannalta). World-luokkaan on myös toteutettu tekoälyn päätöksien määrittävät algoritmit. **Luokalla on yhteys luokkiin Square, Player ja Unit (ja sen alaluokkiin).**

Game
Luokka, joka vastaa vuoronvaihdosta ja kunkin vuoron aikana suoritettujen toimintojen kirjanpidosta (esim. onko pelaaja liikkunut jo vuoronsa aikana). Vastaa myös tekoälyn yksikköjen valinnasta pelin alussa metodin counterpick_ai_units() avulla. Edellä mainittu metodi varmistaa, että tekoäly aina valitsee viisi yksikköä, pelaajan valitsemien yksiköiden määrästä riippumatta. Jokaista pelaajan valitsemaa yksikköä kohti metodi valitsee tekoälylle sellaisen yksikön, joka tehoaa hyvin pelaajan yksikköön. **Luokasta on yhteys luokkiin World, Player ja AI.**

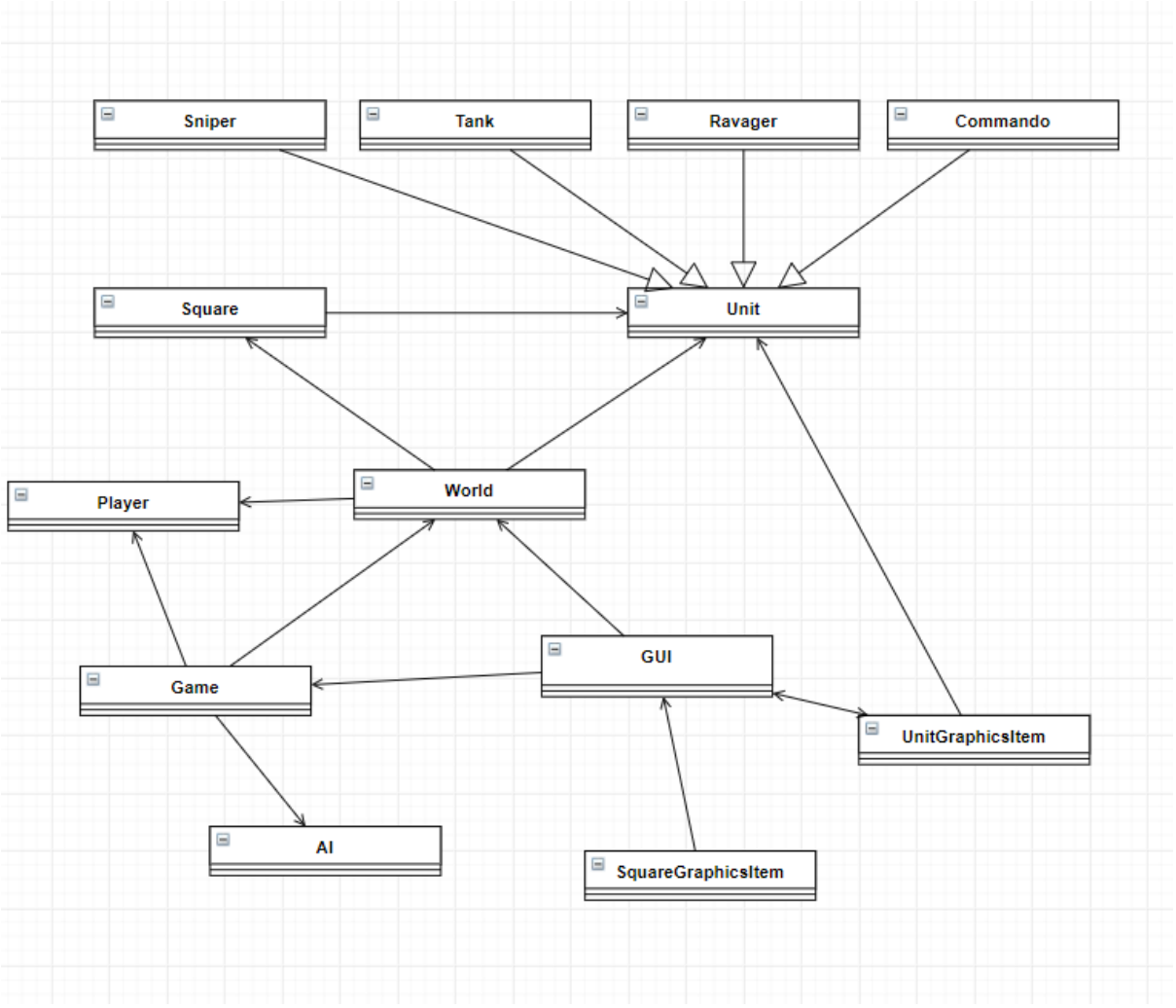
Square
Jokaista ruutua kohti luodaan Square-olio, johon tallennetaan tieto siitä, onko se vapaa, sen sijainti ja lista sen naapuriruuduista (vain ylös/alas/oikea/vasen). Luokka vastaa siis ruutujen tilojen (vapaa/varattu/este) kirjanpidosta. Luokalla on keskeinen metodi update_neighbours(), joka päivittää kyseisen ruudun naapuriruutujen tilan. Tätä metodia kutsutaan luokasta World aina kun pelaaja/tekoäly siirtää yksikköä, tuhoaa toisen yksikön ja vuoron päätteeksi. **Luokasta on yhteys luokkaan Unit (ja sen alaluokkiin).**

Unit (ja sen alaluokat Sniper, Commando, Ravager, Tank)
Luokka Unit on luokka, jonka jokainen yksikkö-olio perii. Se sisältää metodeja, jotka pätevät kaikkiin yksiköihin, niiden ominaisuuksista riippumatta. Luokka vastaa yksikköjen eri tietojen kirjanpidosta ja sisältää metodeja, joilla näitä voidaan noutaa. Erityistapauksissa metodit ovat määritetty uudelleen alaluokassa. Luokka sisältää myös puolustukseen liittyvät metodit, jotka laskevat kunkin yksikön ottaman vahingon ja asettaa mahdollisen verenvuotoefektin. Itse hyökkäysmetodit on toteutettu jokaiselle yksiköllle erikseen, mutta puolustus toimii kaikilla yksiköillä samalla tavalla, joten sen toteutus on Unit-luokassa. Hyökkäysmetodeja kutsutaan luokasta GUI, jossa hyökkäävä ja puolustava yksikkö määritetään. **Itse luokalla ei ole yhteyksiä muihin luokkiin, vaikka siihen on ulkopuolelta yhteyksiä olemassa.**

Player ja AI
Luokat Player ja AI ovat tyhjiä luokkia, joita luulin tarvitsevani projektin alussa. Työn edetessä niiden ainoaksi tarkoitukseksi osoittautui type(Player)-operaatiot, joilla tarkistettiin kuuluuko jokin yksikkö pelaajalle vai tekoälylle. Game-luokassa

type-operaatiota käytetään vuoron määrittämisen. **Luokilla ei ole yhteyksiä muihin luokkiin.**

Ohjelma on hyvin käyttöliittymäpainotteinen, joten suurin osa toiminnoista tapahtuvat luokissa GUI ja World. Kun käyttäjä vuorovaikuttaa käyttöliittymän kanssa, ohjelma päivittää World, Square, Game ja Unit-luokkien tietoa. Tämän tiedon päivityttyä päivitetään käyttöliittymässä näkyvä tieto. Ongelmien erottelu jakautuu käyttöliittymään ja taustalla tapahtuvaan tietoon ja näiden kahden synkronointiin. Player ja AI-luokkien luominen oli virhe. Niistä ei ole mitään hyötyä. Sen sijaan type(Player)-käskyjen sijaan olisi ollut järkevämpää käyttää esimerkiksi parametreja 0 ja 1 pelaajan ja tekoälyn tunnistamiseen. Ohjelma eteni kuitenkin niin pitkälle, että näiden korjaaminen olisi ollut vaivalloista.



Yksinkertainen UML-kaavio luokkien välisistä yhteyksistä

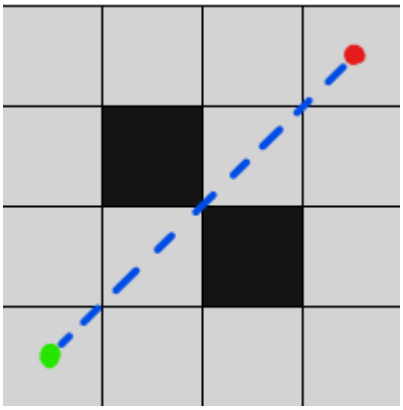
6. Algoritmit

Kun pelaaja yrittää liikuttaa yksikköä tiettyyn ruutuun, ohjelma laskee lyhimmän reitin A* Star-algoritmilla, joka löytyy tiedostosta a_star.py. Algoritmi saa parametrit grid (kentän 2D-taulukko), start, end ja AI (True/False). Parametrit start ja end ovat alku- ja loppupisteen (x,y) koordinaatit. A* Star on yksi maailman tehokkaimmista lyhimmän reitin etsivistä algoritmeista, koska se käyttää heuristista funktiota h(n) reitin pituuden minimoimiseen seuraavan kaavan perusteella:

$$f(n) = g(n) + h(n)$$

$h(n)$ on arvioitu etäisyys solmusta n loppupisteeseen. Koska tässä pelissä ei voi liikkua diagonaalisesti, $h(n)$ lasketaan Manhattan distance-kaavalla: $|\Delta x| + |\Delta y|$ eikä se täten yliarvioi missään tapauksessa etäisyyttä loppupisteeseen.
g(n) on nopein reitti alkupisteestä solmuun n. Koska f(n) on edellä mainittujen summa, se on alku- ja loppupisteen etäisyyksien arvio. Algoritmi on tehokas, koska se ei kulje väärään suuntaan, koska tällöin h(n) kasvaisi. Toinen toteutustapa olisi jokin toinen verkkoalgoritmi, kuten BFS tai DFS, mutta A* on tehokkaampi, enkä ollut oppinut siitä etukäteen Tietorakenteet ja algoritmit -kurssilla.
Toteutin algoritmin siten, että se palauttaa eri arvot riippuen parametrista AI. Kun AI on False, algoritmi palauttaa reitin pituuden tai INF jos reittiä ei ole. Tätä käytetään, kun käyttäjä yrittää liikuttaa yksikköä ruudusta start ruutuun end. Kun algoritmia kutsutaan tekoälyn liikkumisalgoritmista, käytetään AI arvoa True, jolloin algoritmi palauttaa listan lasketun reitin ruutu-olioista ja True tai None ja False, jos reittiä ei ole olemassa. Näin tekoäly voi tarkastella listaa reitin ruuduista ja valita siitä sen ruudun, joka on niin lähellä vihollista kuin sen liikkumiskyky sallii.

Yksikköjen ampumista varten toteutin metodin line_of_sight(), joka tarkistaa olisiko mahdollista ampua toista yksikköä osumatta esteeseen. Tätä varten käytin algoritmia nimeltä “Bresenham’s line algorithm”, jota parantelin hieman. Algoritmi on yksinkertainen: Ensin alku- ja loppupisteen väliin piirretään suora kaavalla $y = ax + b$. Sitten edetään alkupisteestä yhden x-arvon välein kohti loppupistettä ja lasketaan sitä vastaava y-arvo, joka pyöristetään lähimpään kokonaislukuun (koska taulukon indeksit ovat vain kokonaislukuja). Jos mikään ruutu suoralla ei ole seinä, näköyhteys on olemassa (tässä oletetaan, että yksikköjen ohi voi ampua). Algoritmilla on kuitenkin heikkous - se ei tarkastele ruutujen naapuriruutuja lainkaan, jolloin saattaa syntyä tilanne, jossa yksikkö ampuu seinän läpi:



Muutin siis algoritmia siten, että se ottaa huomioon kaikki neljä mahdollista luodin etenemissuuntaa ja tarkastelee niiden kannalta merkittäviä naapuriruutuja. Muita line of sight-algoritmeja on olemassa (linkki viitteissä), mutta niistä yksinkertaisin ja intuitiivisin on Bresenham's line algorithm, joten päätin toteuttaa sen.

Tekoälyn toiminta koostuu kahdesta kokonaisuudesta: liikkuminen ja hyökkääminen. Pelin monimutkaisten sääntöjen vuoksi oli haastavaa käyttää jo tunnettuja tekoälyalgoritmeja. Onko järkevämpää hyökätä ensin ja sitten vasta liikkua, vai toisinpäin? Vaihtoehtoja on paljon, joten tekoälyn toiminta perustuu täysin itse kehitettyyn algoritmiin. Yksinkertaisuuden vuoksi tekoäly aina liikuttaa yhtä yksikköä ensin ja sitten hyökkää jos mahdollista.

Liikkuminen

Tekoäly aina liikkuu ensin ja sitten hyökkää. Liikkumisalgoritmi on toteutettu tiedoston world.py luokan World metodiin get_best_move(). Ensin käydään läpi kaikki tekoälyn yksiköt ja tarkistetaan voivatko ne hyökätä minkä tahansa vihollisen yksikön kimppuun. Jos tämä on mahdollista, niin kyseistä yksikköä ei liikuteta. Tämä takaa sen, että joka vuorolla tekoäly liikuttaa ainakin yhtä yksikköä parempaan sijaintiin, eikä tyydy siihen, että yksi yksikkö "tekee kaiken työn".
Ensisijaisen karsinnan jälkeen etsitään jokaisen jäljelle jääneen yksikön lähin vihollinen ja reitti, jota pitkin yksikkö pääsisi ruudulle, joka on mahdollisimman lähellä kyseistä vihollista. Jokaista yksikköä kohti tallennetaan tämä ruutu. Sitten tarkistetaan, mikä yksikkö pystyisi hyökkäämään jonkun vihollisen kimppuun edellä mainitusta ruudusta. Jos tällainen yksikkö löytyy, liikutetaan sitä. Jos mikään yksikkö ei kykene vuorovaikuttamaan vihollisen kanssa liikkumisen jälkeen, liikutetaan mitä tahansa yksikköä.

Liikkumisen pseudokoodi:

```
moves = []
for unit in ai_units:
   can_att = False
   for enemy in player_units:
       if can_attack(unit, enemy):
           can_att = True
   if can_att:  # If unit can attack an enemy, it shouldn't
move
       continue

   square, enemy = move_closer(unit)
   if square is not None:  # If unit can move closer
       moves.append((unit, square)) # remember the unit,
square and enemy

for move in moves:
 // Jos yksikkö pystyy hyökätä liikkumisen jälkeen, sitä
kannattaa liikuttaa
   if can_attack_from_square(move[0], move[1]):
       return move[0], move[1]

if len(moves) > 0:
// Jos mikään yksikkö ei voi liikkumisen jälkeen hyökätä,
liikutetaan listan ensimmäistä yksikköä
   return moves[0][0], moves[0][1]

return None, None // Jos liikkuminen on mahdotonta tai turhaa
```

Hyökkääminen

Hyökkäämistä varten käytin tekoälytyyppiä nimeltä ‘Utility System’, joka perustuu eri toiminnoista ansaittujen pisteiden maksimointiin. Esimerkiksi, jos yksikkö pystyy vahingoittamaan vihollista, annetaan sille pieni määrä pisteitä. Jos yksikkö pystyy vahingoittamaan jo vahingoittunutta vihollista, annetaan enemmän pisteitä ja jos yksikkö pystyy tuhoamaan vihollisen yksikön, se toteutetaan.
Hyökkäämisalgoritmi on toteutettu tiedoston world.py luokan World metodiin ai_attack().
Jokainen toiminto käydään läpi ja lopulta suurimman pistemäärän saanut toiminto suoritetaan. Pisteitä saa myös sitä enemmän, mitä enemmän vahinkoa yksikkö voi tehdä vihollisen yksikköön. Näin vältetään tilanteet, joissa tarkka-ampuja (Sniper) hyökkäisi tankin kimppuun esimerkiksi kommandon sijaan.

Hyökkäysalgoritmissa käydään läpi jokainen mahdollinen hyökkäys. Jokainen hyökkäys pisteytetään metodin best_attack() avulla. Metodi pisteyttää seuraavat kriteerit:

- Onko vihollinen jo ennaltaan vahingoittunut? +10 p
- Onko hyökkäävä yksikkö tehokas vihollista vastaan? +10 p
- + Yksikön aiheuttaman vahingon odotusarvon verran pisteitä

Itse hyökkääminen tapahtuu unit-tiedostoissa, joissa hyökkääjästä ja puolustajasta luodaan kopio-olion kirjaston copy metodilla deepcopy(). Näitä kopioyksiköitä käytetään simuloimaan kaikki hyökkääjän hyökkäystyypit, joista valitaan suurimman vahingon aiheuttanut. Tässä otetaan huomioon aseiden jäähtymisajat, eli jos esimerkiksi tankin kanuuna on käyttökelvoton, kanuunalla hyökkääminen ei aiheuta vahinkoa, jolloin sitä ei valita.

Hyökkäämisalgoritmin pseudokoodi:

```
score = 0
attacks = {}

for unit in ai_units:
   for enemy in player_units:

       if can_attack(unit, enemy): // Jos unit voi
hyökätä enemy:n kimppuun, edetään

           if can_kill(unit, enemy): // Jos unit voi tappaa
enemyn, tehdään se
               return unit, enemy

           temp_score = best_attack(unit, enemy) // Palauttaa
hyökkäyksen pisteytyksen, lopulta parhaan pistemäärän saanut
hyökkäys toteutetaan

           if temp_score > score:
               score = temp_score
           attacks[temp_score] = (unit, enemy)

if len(attacks) > 0:
   return attacks[score]

return None, None  // Jos ei voi hyökätä, palautetaan None
```

Tekoälyn ei ole tarkoitus olla täydellinen. Se tekee virheitä esimerkiksi yksikön aiheuttaman vahingon odotusarvon laskemisessa, koska se ei ota huomioon tiettyjen yksiköiden aseiden jäähtymisaikaa. Näin ollen se saattaa päättää hyökätä kommandolla tankin kimppuun ja huomata “liian myöhään”, että se ei voi käyttää panssarintuhoamisasettaan. Tämä on kuitenkin osa, joka tekee pelin kulusta hauskemman, eivätkä nämä virheen vaikuta peliin ratkaisevasti.

Alunperin halusin toteuttaa tunnetun tekoälyalgoritmin nimeltä Minimax, jota käytetään mm. shakkitekoälyissä, mutta pelin monimutkaisen luonteen vuoksi jouduin tyytymään itse kehittämääni algoritmiin. Kirjoittamani tekoäly kuitenkin toimii hyvin ja tekee erittäin järkeviä päätöksiä.

7. Tietorakenteet

Ohjelmassa ei käytetä monimutkaisia tietorakenteita. Siinä pääosin käytetään muuttuvia tietorakenteita jatkuvasti muuttuvan tiedon vuoksi. Tieto tallennetaan listoihin, joissa sitä on helppo käsitellä sekä monikkoihin. Maailman generointiin käytetään kaksiulotteista listaa yksinkertaisuuden ja helppouden vuoksi. Siitä voi esimerkiksi helposti noutaa indeksien avulla tiettyjä square-olioita. Yksiköiden ja niiden graphics itemien oliot tallennetaan listoihin luokkien World ja GUI attribuutteihin, vastaavasti. Jokaisen ruudun naapuriruudut tallennetaan myös listoihin, sillä niiden tilat muuttuvat jatkuvasti. A* Star-reittialgoritmissa käytetään prioriteettijonoa ja sanakirjaa algoritmin luonteen vuoksi. Sanakirjaa käytetään myös tekoälyn algoritmissa, jossa kukin siirto saa tietyn pisteytyksen, joka toimii avaimena. Suurin avain on helppo löytää ja siten sitä vastaava siirto. Muihin vaihtoehtoihin olisi voinut kuulua esim. puurakenteet, mutta käsiteltävän data määrä on niin pieni, etten nähnyt tarpeelliseksi ottaa erityisesti huomioon puurakenteiden hyötyjä.

8. Tiedostot

Ohjelma käsittelee pelkkiä python (py) tiedostoja. Kaikki tarvittavat tiedostot löytyvät src-kansiosta ja ne ovat tarpeellisia ohjelman ajamisen kannalta. Mitään muita tiedostoja ei tarvita.

9. Testaus

Ohjelman testaus ei vastannut suunnitelmassa esitettyä. Käyttöliittymän testaaminen yksikkötesteillä osoittautui niin hankalaksi, että maailman ja esteiden generointi testattiin yksikkötesteillä ja käyttöliittymään littyvät ongelmat manuaalisesti

print-käskyillä ja toistoilla. Esimerkiksi yksikköjen ja ruutujen sijaintien testauksessa print-käskyllä kykenin tulostamaan klikkaamani ruudun/yksikön koordinaatit. Reittialgoritmin testauksessa reitin koordinaattien tulostaminen osoitti reitin johdonmukaisuuden ja oikeaoppisuuden.
Jokaisen kokonaisuuden luomisen jälkeen kokonaisuus testattiin. Jos siitä löytyi jälkikäteen bugeja, ne korjattiin.

Yksikkötesteihin kuuluvat metodit:
test_world_generation() joka testaa kentän mittakaavojen oikeaoppisuuden
test_obstacles() joka testaa esteiden koordinaatteja siemenluvulla 100.

Testaussuunnitelma oli hyvä, mutta ohjelman edetessä vastaan tuli monia muita ongelmia, joita piti testata (esimerkiksi Bresenhamin algoritmin erityistapausta en olisi saanut selville testaamatta algoritmia useaan otteeseen).

10. Ohjelman tunnetut puutteet ja viat

Luokat Player ja AI ovat täysin turhia, kuten aiemmin mainittu. Luulin alussa tarvitsevani niitä, mutta huomasin, että itse luokat olisin voinut korvata pelkillä parametreilla 0 ja 1. Nyt tilanteen korjaaminen vaatisi kaikkien tiedostojen läpikäymistä systemaattisesti.

Olisin voinut kirjoittaa koodia johdonmukaisemmin. Huomasin jälkikäteen, että yksikön sijainnin palauttava metodi palauttaa sijainnin monikossa (x,y) kun taas ruudun sijainnin palauttava metodi palauttaa kaksi arvoa x, y. Tällainen epäjohdonmukaisuus osoittautui ärsyttäväksi ohjelman edetessä. Lisäksi, joissakin metodeissa vihollisyksikkö on nimetty parametrilla ‘enemy’ ja muissa ‘e_unit’ tai ‘p_unit’. Tulevaisuudessa pidän mielessä sen, että koodin johdonmukaisuus on tärkeää.

Ohjelman tapa valita tekoälyn yksiköt riippuu tämänhetkisten yksiköiden vahvuuksista ja heikkouksista. Esimerkiksi jos pelaaja asettaa tankin kentälle, tekoäly valitsee joko Commandon tai Ravagerin. Jos peliin lisättäisiin uusia yksiköitä, jotka osoittautuisivat vielä paremmaksi valinnaksi, tekoäly ei koskaan valitsisi niitä, sillä se on ohjelmoitu valitsemaan joko Commandon tai Ravagerin. Sama ongelma toistuu World-luokan metodissa counters(), joka antaa tekoälylle lisäpisteitä jos sen hyökkäävä yksikkö on tehokas puolustavaa yksikköä vastaan. Yritin ratkaista tämän ongelman lisäämällä jokaiselle yksiköllе lista-attribuutin counters, johon tallennettaisiin kaikki yksikkötyypit, jota vastaan kyseinen yksikkö on tehokas. Tässä vastaan tuli kuitenkin import-käskyjen kehämäisyys. Ongelman voisi ratkaista esimerkiksi simuloimalla hyökkäyksiä eri yksiköillä ja määrittää onko yksikkö tehokas

puolustavaa yksikköä vastaan tarkastelemalla sen suhteellista elämäpisteiden menetystä.

Luokassa GUI määritelty metodi end_current_turn() on toteutettu puutteellisesti. Siinä käydään läpi pelaajan ja tekoälyn yksiköt erikseen, jolloin samat käskyt on kirjoitettu kahdesti kahteen erilliseen for-käskyyn. Tämän voisi ratkaista esimerkiksi luomalla attribuutin all_units_graphics_items, joka sisältäisi kaikki kentällä olevien yksiköiden UnitsGraphicsItem-oliot. Näin ratkaistaisiin koodin toisteisuusongelma.

World-luokassa on toteutettu metodi, joka asettaa yksiköt kentälle siten, että ne sijoittuvat mahdollisimman keskelle omalla rivillään. Tämä metodi ei kuitenkaan ota huomioon sitä, jos käyttäjä päättää ajaa ohjelman kentällä, jonka koko on pienempi kuin 5x5. Tällöin ohjelma kaatuu index out of range erroriin. HUOM, peli on kuitenkin tarkoitettu pelattavaksi 10x10 kokoisella kentällä. Sen pelaaminen pienemmällä kentällä olisi yhtä järjetöntä kuin shakin pelaaminen pienemmällä pelilaudalla. Jos jatkaisin projektia, korjaisin tämän ongelman siten, että ohjelma ei kaatuisi. Asettaisin siis yksikköjä enintään kentän leveyden salliman määrän.

11.3 parasta ja 3 heikointa kohtaa

3 parasta:
A* Star reittialgoritmin opettelu ja toteuttaminen oli vaikeaa, mutta palkitsevaa. Olisin voinut toteuttaa ennalta tutumman algoritmin, mutta halusin oppia uutta. Kun löysin kyseisen algoritmin internetistä päätin, että haluan toteuttaa sen ohjelmassani. Vaikka aika-kompleksisuudella ei ole merkittävää vaikutusta 10x10 kentällä, A* Star on huomattavasti tehokkaampi kuin muut vaihtoehdot suuremmissa mittakaavoissa.

Line of sight-algoritmi lisää pelin realistisuutta ja tekee siitä mielenkiintoisemman. Olen erityisen ylpeä myös omasta muutoksesta, jonka tein siihen ratkaistakseni ongelman, jossa seinän läpi pystyi ampumaan.

Tekoäly ei onnistunut niin hyvin kuin olisin halunnut, mutta olen silti hyvin ylpeä siitä. Sen kirjoittaminen osoittautui suureksi haasteeksi, koska peli on hyvin monimutkainen. Yksikköjen aiheuttama vahinko ei ole vakio, eräät yksiköt aiheuttavat verenvuotoefektin, yhdellä yksiköllä voi liikkua, toisella voi hyökätä sekä näkyvyys ja ampumakantaman aspektit on otettava huomioon. Toisin kuin perinteiset strategiapelit, kuten shakki, minun strategiapelini ei ole ns. täydellisen informaation strategiapeli, missä seuraavan vuoron tilanne voidaan päätellä yhdellä siirrolla. Tämän takia strategiapeleihin kehitetyt algoritmit kuten Minimax ja sen lisäominaisuus Alpha-Beta pruning osoittautuivat huonoksi valinnaksi. Kirjoittamani tekoäly tekee kuitenkin usein hyviä päätöksiä ja voittaa hämmästyttävän helposti, jos pelaaja ei ole tarkkana.

3 huonointa:

Ohjelmassani olisin voinut miettiä enemmän sitä, kuinka helppoa valmiiseen peliin olisi lisätä ominaisuuksia. Kuten edellä mainitsin, tekoälyn yksikkö- ja hyökkäysvalinnat perustuvat jo olemassa oleviin yksiköihin. Täten src-kansioon uuden yksikkötyypin tiedoston lisääminen ei riittäisi, vaan jouduttaisiin tekemään muutoksia monen luokan koodiin. (mm. World, UnitsGraphicsItem ja Game). Yritin kuitenkin minimoida tarvittavien muutosten määrän.

Luokat Player ja AI ovat turhia. Poistaisin ne, jos minulla olisi enemmän aikaa, mutta tämä vaatisi muutoksia monessa eri luokassa. Lisäksi muutamassa kohdassa kuten GUI-luokan metodissa end_current_turn() esiintyy hieman koodin toisteisuutta.

Koodin epäjohdonmukaisuus parametrien nimeämisessä ja eri luokkien metodeissa on myös kehitettävä osa-alue. Tulevaisuudessa pidän oppimani asiat mielessä.

12. Poikkeamat suunnitelmasta

Suunnitelma auttoi erittäin paljon ohjelman luomisen alussa. Projektin edetessä siitä oli vähemmän ja vähemmän hyötyä. Huomasin toteuttaneeni testaussuunnitelman ja luokat eri tavalla kuin suunnitelmassa. Suunnitelma laadittiin hyvin aikaisin, ilman mitään selkeää visiota, joten on loogista, että lopputyö ei täysin noudata sitä. Testaukset suoritin pääasiassa manuaalisesti käyttöliittymän aiheuttaneiden haasteiden vuoksi, kuten selitin kohdassa 9. Player ja AI-luokat osoittautuivat turhiksi. Päätin käyttää perintää yksiköille, koska niillä osoittautui olevan enemmän yhteistä kuin luulin. Lisäksi loin jokaista ruutua varten SquareGraphicsItem-olion, jonka avulla sain kunkin ruudun tarkan koordinaatin kentällä.

Projektin aikataulu noudatti aikalailla suunnitelmaa tekoälyn ja reittialgoritmien kirjoittamista lukuunottamatta. Niihin kului huomattavasti enemmän aikaa kuin suunnittelin. Tekoälyn kirjoittamiseen kului aikaa, koska peli on monimutkainen ja monta asiaa on otettava huomioon. Reittialgoritmin kirjoittamisessa meni aikaa, koska siihen piti ensin perehtyä ja sitten ratkaista ruutujen naapuriruutujen laskemisongelma. Kaiken lisäksi piti tajuta, että kaksiulotteisessa taulukossa x- ja y-akselit ovat käyttöliittymän sceneen verrattuna toisinpäin.

Toteutusjärjestys oli osittain järkevä. Start game näppäin ohjelmoitiin heti yksiköiden lisäämisominaisuuden jälkeen. Sitten toteutin liikkumisominaisuuden, hyökkäykset, line of sight, tekoälyn ja pelin lopettamisen.

13. Toteutunut työjärjestys ja aikataulu

23.2.2021 - main, luokat Square, World, Game, Player, AI luotu
24.2.2021 - esteiden generointi ja testaaminen (test.py), GUI luotu (kenttä näkyy ajaessa main.py:n), yksikköjen tiedostot luotu

25.2.2021 - unit-tiedosto luotu (ei suunnitelman mukainen) perintää varten, luokka UnitGraphicsItem luotu

26.2.2021 - ‘start game’ nappäin (ei suunnitelman mukaiseen aikaan), statusbar
27.2.2021 - luokka SquareGraphicsItem luotu (ei suunnitelman mukainen)
1.3.2021 - A* Star reittialgoritmi
2.3.2021 - Vuoron toimintojen kirjanpitometodit ‘unselect unit’-ominaisuus
26.3.2021 - Hyökkäysvaihtoehtojen lisääminen
27.3.2021 - Hyökkäysmetodien toteutus, vuoron vaihtuminen, bleeding damage
28.3.2021 - Hyökkäysmetodien viimeistely
29.3.2021 - Lisätty hyökkäysmetodi GUI:hin
7.4.2021 - Line of sight algoritmi
16.4.2021 - Tekoälyn aloittaminen, reittialgoritmin korjaaminen
17.4.2021 - Tekoälyn hyökkäminen
20.4.2021 - Ajastimen lisäys
28.4.2021 - Reittialgoritmin muuttaminen, tekoälyn liikealgoritmin viimeistely
30.4.2021 - Tekoälyn hyökkäysalgoritmin korjaaminen ja viimeistely

Nämä päivämäärät eivät ole ainoat, jolloin työstin projektia. Nämä ovat vain committien päivämääriä. Niiden välissä on tehty monen päivän ajan pieniä ohjelmointisessioita, joita ei ole kaikkia dokumentoitu. **Toteutustaso on keskivaikea**. Aikaa kului reilusti enemmän kuin suunnitelmassa, etenkin reittialgoritmin ja tekoälyn toteuttamiseen.

14. Arvio lopputuloksesta

Ohjelma on yksinkertainen ja tehokas. Sen pystyy ajamaan hitaillakin tietokoneilla ongelmitta, koska sen käyttäminen ei vaadi suuria määriä resursseja. Toteuttaessani ohjelmaa olen miettinyt eri algoritmien aikakompleksisuutta ja sen minimointia, vaikka käsittelemäni tiedon koko ei olekaan niin suuri, että sillä olisi ratkaisevaa

merkitystä. Oleellisia puutteita ohjelmassa ei ole: Se toimii niin kuin pitää eikä kaadu, kunhan sen ajaa oikealla kentän koolla. Pitkän kantaman yksiköt pysyttelevät kaukana, jos vain mahdollista ja lyhyen kantaman yksiköt (kuten Ravager) ryntäävät kohti vihollista antaen pelille persoonallista vaihtelua. Ohjelmassa on toteutettu algoritmeja, joista on hyötyä jatkossa (esim line of sight ja A* Star) ja niiden toteuttaminen on kehittänyt ohjelmointitaitojani huomattavasti. Ohjelmassa on käytetty kurssilla opetettua perintää, yksikkötestausta sekä PyQt5-käyttöliittymää. Ohjelma sisältää kaikki keskivaikean toteutustason vaatimukset ja vähän enemmänkin.

Näistä huolimatta ohjelmassa on paljon parannettavaa, kuten koodin toisteisuus tai sen kaatuminen ajettaessa pienellä kenttäkoolla. Suurin ongelma on ohjelman laajennettavuus tulevaisuudessa, mitä en ottanut huomioon niin paljon kuin olisi pitänyt. Projekti on ollut pitkä oppimiskokemus, jonka aikana tietoa ja osaamista on kertynyt entistä enemmän. Näin ollen huomasin useaan otteeseen vasta jälkikäteen, että olisin voinut toteuttaa joitakin asioita paremmin.
Tietorakenteiden valinnan kanssa minulla ei ole katumuksia. Koodi olisi voinut olla johdonmukaisempaa ja luokkavalinnat voisivat olla parempia (AI ja Player).
Ohjelman laajennettavuus ei ole paras, mutta ei huonoinkaan. Pienten muutosten jälkeen uusien yksiköiden lisääminen peliin olisi hyvin helppoa. Tällä hetkellä jos haluttaisiin lisätä uusi yksikkö, sitä varten pitäisi luoda ulkonäkö (UnitsGraphicsItem) ja tehdä muutama muutos tekoälyn algoritmeissa.

15. Viitteet

Kurssikirja:
CS-A1121 Ohjelmoinnin peruskurssi Y2 | A+
https://docs.python.org/3/
https://docs.python.org/3/library/
https://docs.python.org/3/library/copy.html
https://docs.python.org/3/library/random.html
https://docs.python.org/3/library/timeit.html
https://docs.python.org/3/library/queue.html

PyQt5:
https://www.riverbankcomputing.com/static/Docs/PyQt5/
https://doc.qt.io/qt-5/

Line of sight ja tekoäly:
https://en.wikipedia.org/wiki/Utility_system
https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php

http://www.roguebasin.com/index.php?title=Line_of_sight
https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
https://en.wikipedia.org/wiki/Minimax
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Reittialgoritmit:
https://en.wikipedia.org/wiki/Taxicab_geometry
https://en.wikipedia.org/wiki/A*_search_algorithm

16. Liitteet

Toteutettu kohdassa 3. Käyttöohjeet