CSE-A1121 Projektityön dokumentti, Strategiapeli ”Disco Knights”

1. Henkilötiedot

Nimi: Roope Palomäki
Koulutusohjelma: KTM Finance
Päiväys: 10.5.2015

Opiskelijanumero: 240938
Vuosikurssi: 2010

2. Yleiskuvaus

Disco Knights on vuoropohjainen strategiapeli, jota voi halutessaan pelata tietokonevastustajaa vastaan. Pelissä hahmot liikkuvat ruudukolla / kartalla, ja voivat yhden vuoron aikana sekä liikkua että käyttää jotain toimintoa. Vuoron voi myös lopettaa tekemättä mitään. Hahmot liikkuvat kohtisuoraan ruutuihin nähden, ja kartalla on sekä erilaisia ruututyyppejä että esineitä/objekteja, joiden läpi ei välttämättä pysty liikkumaan. Ruututyyppejä ovat esimerkiksi vesi ja ruoho, ja objekteja kivi sekä puu. Hahmojen toiminnot saattavat joko vahingoittaa tai parantaa toista hahmoa, vaikuttaa toisen hahmon toimintojen vahvuuteen tai tainnuttaa kohteena olevan hahmon hetkeksi.

Karttoja ja hahmoja pystyy muokkaamaan ja lisäämään asetustiedostojen kautta. Karttatiedostossa pystyy määrittämään ruututyyppejä ja esineitä/objekteja, joista kartan voi sitten rakentaa haluamallaan tavalla. Karttoja voi tehdä haluamansa määrän. Hahmotiedostossa pystyy määrittämään hahmojen ominaisuudet ja toiminnot, joita kukin hahmo voi käyttää. Jokaisella hahmolla voi olla maksimissaan kolme eri toimintoa käytössään. Hahmotiedostossa määritetään samoin, onko kukin hahmo tekoälyn kontrolloima vai ei. Näin peliä voi halutessaan pelata joko toista pelaajaa tai sitten tekoälyä vastaan.

Asetustiedostojen avulla on myös helppo muuttaa pelin ulkonäköä, sillä elementteihin sisältyy määrityksenä myös kuvatiedostot, joista ne piirretään. Esimerkiksi uusia ruututyyppejä tai objekteja onkin helppo lisätä vain piirtämällä niille kuva ja lisäämällä asetustiedostoon, jonka jälkeen niitä voi käyttää kartan rakentamiseen. Sama pätee myös hahmoille. Jokaiselle toiminnolle on myös mahdollista asettaa ääniefekti, joka toistetaan kun toimintoa käytetään. Peliä on siis helppo muokata ja periaatteessa siitä voisi melko helposti tehdä toisen saman tyyppisen pelin eri teemalla.

Peli osaa ohjastaa hahmoja haluttaessa tekoälyn avulla. Tekoälyssä on kaksi osuutta: kohteen valitseminen ja toiminnon valitseminen. Kohde valitaan vihollisista perustuen hahmojen sijaintiin ja jäljellä olevaan ”terveyteen” niin, että heikommassa kunnossa ja lähempänä olevat hahmot tulevat valituksi helpommin. Käytettävä toiminto taas valitaan siten, että useimmiten valitaan toiminto, jolla on suurin vaikutus esimerkiksi tehdyn vahingon osalta. Toimintojen valinnassa on mukana suhteellisen pieni satunnaisuustekijä, jolloin tekoäly on suurimman osan ajasta rationaalinen, mutta käyttäytyy joskus epärationaalisesti. Tekoäly osaa myös ottaa huomioon mahdolliset oman joukkueen hahmoihin kohdistuvat toiminnot, ja saattaa näin ollen joskus esimerkiksi parantaa omia hahmoja. Halutessaan yhtään hahmoa ei tarvitse asettaa tekoälyn kontrolloimaksi, jolloin peliä voi pelata toisen pelaajan kanssa vuorotellen.

Itse pelin lisäksi ohjelmaan on luotu valikkosysteemi, josta valitaan pelattava kartta ja josta voi esimerkiksi muuttaa ääniasetuksia tai siirtyä koko näytön tilaan ja takaisin ikkunatilaan. Valikot mukautuvat esimerkiksi valittavien vaihtoehtojen ja määriteltyjen karttojen määrän mukaan ja näyttävät esikatselun kartoista ennen pelattavan kartan valintaa. Valikkojärjestelmä on suunnitelmaan nähden uutta.

Suunnitelmaan nähden uutta on myös ohjelman tilan hallintajärjestelmä sekä tapahtumienhallinta. Koko ohjelmaa hallitsee tilanhallintajärjestelmä, joka pitää huolen siirtymisistä valikkotilojen ja pelitilan välillä, ja tapahtumia, kuten hiiren ja näppäimistön painalluksia hallitsevat kunkin tilan tapahtumankäsittelijät. Tämä taustajärjestelmä mahdollistaisi esimerkiksi uusien valikoiden lisäämisen melko pienellä vaivalla. Ohjelman rakenteesta kerrotaan enemmän sille tarkoitetussa osiossa.

Projektin toteutuksessa on tähdätty vaikeusasteeseen ”vaativa”, ja omasta mielestäni se on toteutettu sen tasoisesti.

3. Käyttöohje

Ohjelma käynnistetään ajamalla tiedosto ”discoknights.pyw”.

Jos hahmoja tai karttoja haluaa kustomoida, tulee muutokset tehdä asetustiedostoihin ”map_config.txt” ja ”character_config.txt” ennen ohjelman käynnistämistä. Harvemmin muutettavia asioita voi myös muuttaa ”constants.py” -tiedostossa.

Peli koostuu valikoista ja itse pelitilasta, ja sitä käytetään/pelataan hiiren avulla. Se käynnistyy päävalikkoon, josta voi halutessaan siirtyä toisiin valikkoihin ja esim. muuttaa ääniasetuksia (asettaa ovatko musiikit ja ääniefektit päällä) tai siirtyä koko näytön tilaan. Valitessaan ”New Game” pelaajalle esitetään saatavilla olevat kartat, ja peli alkaa, kun kartta valitaan.

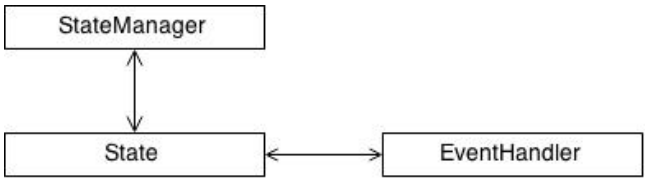
Kun hahmolle tulee vuoro, kartalla korostetaan ruudut, joihin hahmo pystyy liikkumaan tämän vuoron aikana. Liikkua voi klikkaamalla kohderuutua, ja toimintoja voi käyttää valitsemalla ensin haluamansa toiminnon vasemmassa alanurkassa olevasta valitsimesta, ja sen jälkeen klikkaamalla kohderuutua/-hahmoa. Kun toiminto on käytetty, vuoro loppuu automaattisesti. Vuoron voi myös lopettaa klikkaamalla ”End Turn” nappia oikeassa alanurkassa, jos esimerkiksi vihollisen hahmoja ei ole riittävän lähellä tai ei halua tehdä mitään liikettä. Vuorojenhallinta osaa automaattisesti kontrolloida tekoälyhahmoja, kun niiden vuoro tulee. Peli myös ohjeistaa liikuttamaan hahmoa tai suorittamaan toiminnon, kun jokin pelaajan hahmo saa vuoron.

Ikkunan ylälaidassa näkyy pelin aikana tieto hahmojen tilasta, eli kuinka paljon hahmoilla on elämää jäljellä, onko hahmo hengissä vai ei, minkä hahmon vuoro on tällä hetkellä, ja onko hahmo tainnuksissa.

Kesken pelin taukovalikkoon pääsee normaaliin tapaan Esc-näppäimellä, ja peliä voi sieltä jatkaa tai aloittaa kokonaan uuden pelin, tai myös halutessaan sulkea ohjelman.
Kun toinen joukkue voittaa pelin, näytetään kuka pelin voitti, ja pelaaja voi palata takaisin päävalikkoon, josta voi jälleen aloittaa uuden pelin tai poistua ohjelmasta.

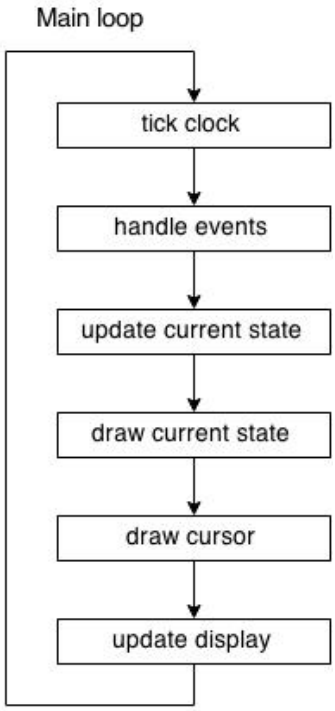
4. Ohjelman rakenne

Ohjelma rakentuu yksinkertaistettuna seuraavalla tavalla. Se rakentuu tiloista, State, joiden välillä liikkumista kontrolloi StateManager. Jokaiseen tilaan liittyy EventHandler, joka käsittelee tapahtumat, kuten hiiren klikkaukset ja näppäimistön painallukset sekä tekoälyn laukaisemat tapahtumat.



Jokaisella valikolla ja itse pelillä on oma State -luokan tai sen perillisen olionsa, ja jokaiseen State -olioon liittyy EventHandler -luokan tai sen perillisen olio.

Pohjimmiltaan ohjelmaa pyörittää pääsilmukka main-py -tiedostossa. Main-funktiossa alustetaan Pygame ja StateManager, jonka jälkeen aloitetaan pääsilmukka. Siinä päivitetään StateManagerin avulla nykyistä ohjelman tilaa. Silmukan toiminta näyttää konseptina seuraavalta:



StateManager pitää kullakin hetkellä tallessa ainoastaan nykyistä tilaa, jota vaihdetaan tarvittaessa. Tämän sijasta voitaisiin tehdä esimerkiksi pino tiloille, jolloin tilasta poistuttaessa palattaisiin taas pinon toiseksi päällimmäiseen tilaan, ja näin voitaisiin toteuttaa vaikkapa valikoita, jotka leijuvat pelin päällä. Tässä ohjelmassa se ei ole kuitenkaan tarpeen, koska erityisen monimutkaista liikkumista tilojen välillä ei tapahdu, ja voidaan käyttää yksinkertaisempana ratkaisuna vain nykyisen tilan tietoa.

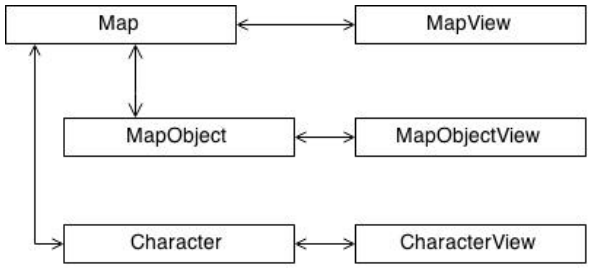
Ensimmäisessä suunnitelmassani ja toteutuksessani käytin rakennetta, jossa lähes kaikki käyttöliittymään liittyvä oli kirjoitettuna main.py -tiedostossa ja suurelta osin main-funktion sisällä. Eri tiloille ei oltu määritelty erityistä rakennetta, vaan liikuttiin useiden silmukoiden välillä pääsilmukan sisällä. Tämä oli erittäin sekava rakenne toteuttaa koko ohjelma, joten suunnittelin uudelleen ja rakensin esitellyn tilajärjestelmän ja erilliset tapahtumankäsittelijät. Sekavuuden lisäksi kaiken sijoittaminen pääsilmukkaan aiheutti myös useissa kohdissa suorituskykyongelmia.

Sen lisäksi, että pääsilmukan liiallinen täyttäminen vaikeuttaa koodin kirjoittamista, lukemista ja ymmärtämistä, on se myös melko kiinteä tapa tehdä asioita, jolloin ohjelman laajentaminen on vaikeaa. Sen sijaan toteutetulla tilasysteemillä ohjelmaan olisi helppo lisätä uusia tiloja ja laajentaa sitä. Iso osa koodista myös soveltuisi nyt myös käytettäväksi toisissa projekteissa.

Ohjelmassa käytettävät tilaluokat ja niiden käyttämät tapahtumankäsittelijäluokat ovat seuraavat. Sulut kuvaavat perintää. Kaikki tilat kuitenkin perivät myös State-luokalta, ja kaikki tapahtumankäsittelijät EventHandler-luokalta.

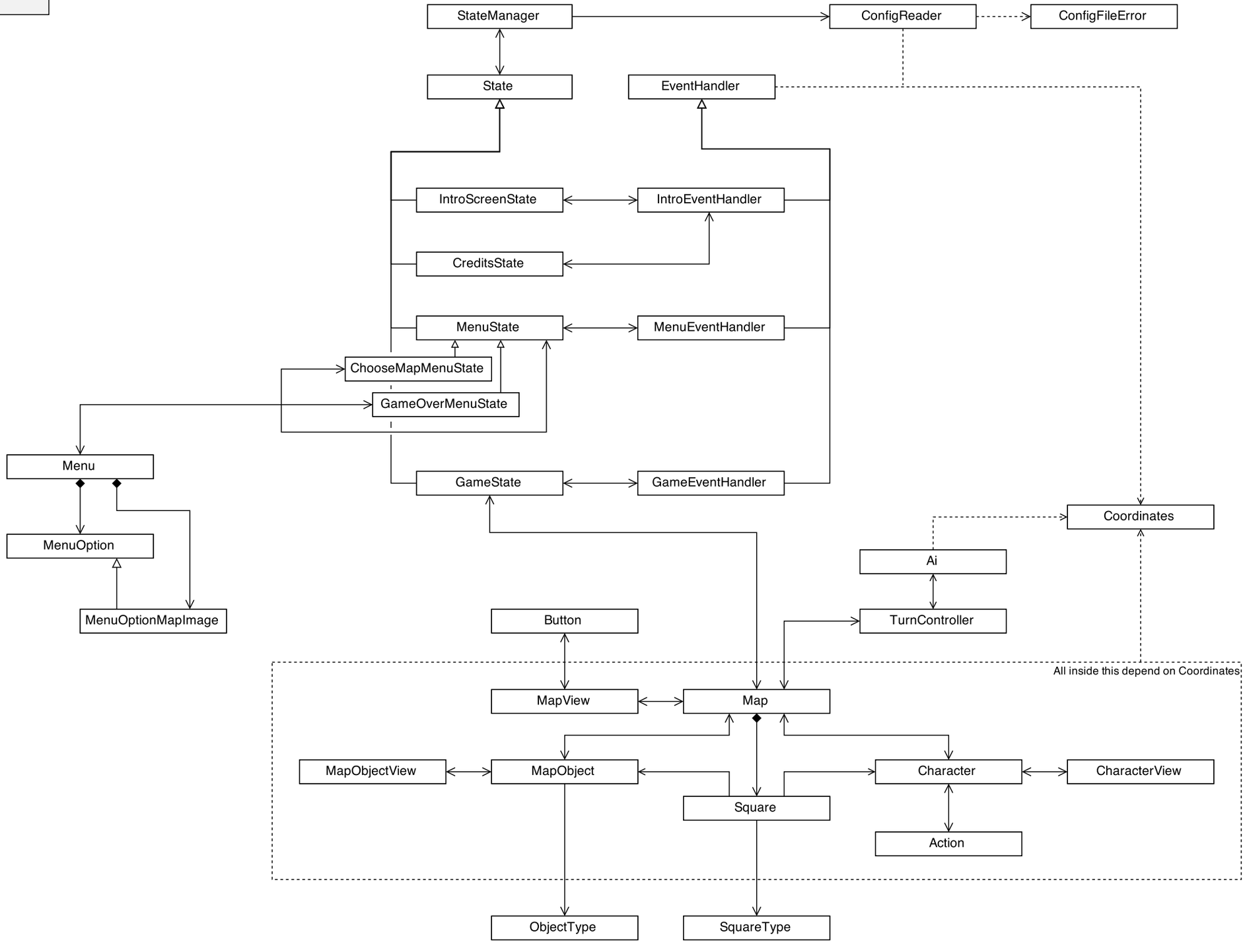
Tila	Tapahtumankäsittelijä
IntroScreenState	IntroEventHandler
CreditsState	IntroEventHandler
MenuState	MenuEventHandler
ChooseMapMenuState(MenuState)	MenuEventHandler
GameOverMenuState(MenuState)	MenuEventHandler
GameState	GameEventHandler

Itse pelin elementit (GameStaten sisällä) pohjautuvat rakenteeseen, jossa on erotettu taustalogiikka käyttöliittymän piirtämisestä. Näin kartalla, objekteilla ja hahmoilla on erikseen myös näkymä, joka hoitaa grafiikoiden päivittämisen ja piirtämisen.



Yllä olevat kuvaajat ovat vain osittaisia havainnollistuksia. Koko ohjelman luokkarakenne on kuvattu alla olevassa kaaviossa. Kuvaajan jälkeen esitellään luokkien sisältö.

Disco Knights class structure
Roope Palomäki
240938



StateManager

StateManager kontrolloi ohjelman tilojen välillä liikkumista ja ylätasolla yhdistää käyttöliittymän pääsilmukan tiloihin ja niiden päivittämiseen.

Alustuksessa asetetaan koko näytön, musiikin ja ääniefektien oletusasetukset, asetetaan muokattu hiiren osoitin, alustetaan ConfigFileReader -olio asetusten lukua varten, kutsutaan metodia, joka alustaa kaikki ohjelman tilat, ja asetetaan käynnistymistilaksi ns. intro screen.

Metodit:

init_states()
Alustaa kaikki tilat, joita käytetään ohjelmassa, ja lisää valikkotiloihin painikkeet.

go_to(state)
Siirtää ohjelman parametrina annettuun tilaan. Nollaa käyttöliittymää varten näytön ja vaihtaa peliin siirryttäessä pelimusiikin sekä tarvittaessa päävalikossa intromusiikin.

go_to_game()
Kutsuu edellistä metodia parametrina pelitila. Tätä metodia tarvitaan erikseen, koska valikkovaihtoehdoille on määritelty funktio ja parametri, jolla sitä kutsutaan jo alussa. Ohjelman käynnistyessä pelitilaksi asetetaan *None*, koska peliä ei ole käynnissä, joten valikkovaihtoehtojen parametreiksi joutuu myös *None*. Tämä sivuvaikutus aiheutuu pohjimmiltaan siitä, että Python on ns. ”pass-by-assignment”.

quit()
Sulkee ohjelman turvallisesti.

reset_screen()
Palauttaa nollatun Pygamen display Surfacen eli nollaa näytön. Ottaa huomioon senhetkisen fullscreen -asetuksen.

toggle_fullscreen()
Liikkuu koko näytön tilaan tai pois sieltä. Muuttaa myös näyttöasetusvalikon tekstin kuvastamaan asetusta.

play_music(file_path, loop=-1, new_volume=None)
Soittaa parametrina annetussa tiedostopolussa olevan äänitiedoston. Toistaa sitä loputtomasti, jos parametria ”loop” ei muuteta oletuksesta. Muuttaa haluttaessa äänenvoimakkuuden parametrina annetuksi.

toggle_music()
Vaihtaa musiikin äänenvoimakkuuden päälle tai pois. Muuttaa myös ääniasetusvalikon tekstin kuvastamaan asetusta.

toggle_sound_effects()
Vaihtaa ääniefektit päälle tai pois. Muuttaa myös ääniasetusvalikon tekstin kuvastamaan asetusta.

State

Yliluokka ohjelman tiloille. Alustuksessa yhdistetään tila StateManageriin ja luodaan sille nollattu näyttö.

Metodit:

handle_events()
Käy läpi Pygamen tapahtumapinoa ja syöttää tapahtumia tilan tapahtumankäsittelijälle.

update()
Nostaa ainoastaan NotImplementedError -virheen, ja varmistaa näin että update() -metodi on toteutettu aliluokissa.

draw()
Nostaa ainoastaan NotImplementedError -virheen, ja varmistaa näin että update() -metodi on toteutettu aliluokissa.

draw_cursor()
Piirtää hiiren osoittimen näytölle hiiren sijainnin mukaan.

IntroScreenState(State)

Tila, johon ohjelma käynnistyy. Alustuksessa luodaan tapahtumankäsittelijä, asetetaan taustakuva, piirretään ”press any key” teksti pinnaksi, pyydetään soittamaan intromusiikki sekä alustetaan ajastin, jonka avulla muutetaan tekstin väriä säännöllisin väliajoin.

Metodit:

toggle_image()
Vaihtaa tekstikuvaa harmaasta valkoiseen tai päinvastoin. Teksti on piirretty Pygamen Surface-olioihin.

update()
Kutsuu edellistä metodia, jos aikaa on kulunut vähintään puoli sekuntia, ja vaihtaa näin tekstin väriä edestakaisin.

draw()
Piirtää taustakuvan ja tekstin näytölle.

CreditsState(State)

Tila, jossa näytetään credits -tekstit. Alustuksessa ladataan taustakuva, asetetaan sama tapahtumankäsittelijä kuin intro screenille, ja kutsutaan metodia, joka piirtää tekstit.

Metodit:

get_credits()
Hakee tekstit oletuksena credits.txt -tiedostosta. Tiedosto on määritetty constants -moduulissa.

draw_credits()
Piirtää edellisen metodin hakeman tekstin Pygamen Surface-oliolle.

reset_position()
Siirtää tekstit näytön alapuolelle.

update()
Siirtää tekstipintaa ylöspäin ja takaisin näytön alapuolelle, jos ne ovat liikkuneet kokonaan näytön yli.

draw()
Piirtää taustakuvan ja tekstipinnan näytölle.

MenuState(State)

Tila, jossa näytetään jokin valikko. Alustus luo Menu-olion, tapahtumankäsittelijän ja lataa taustakuvan.

Metodit:

add_option(text, function, func_parameter=None, greyed=False)
Luo uuden MenuOption-olion ja lisää sen tilan Menu-olioon. Parametreina annetaan teksti, kutsuttava funktio, ja haluttaessa funktiolle parametri ja onko vaihtoehto käytettävissä.

set_rects()
Asettaa valikkovaihtoehtojen sijoittelun näytöllä niiden lukumäärän mukaan.

update()
Kutsuu jokaisen valikkovaihtoehdon update()-metodia.

draw()
Piiirtää taustakuvan ja valikkovaihtoehdot.

ChooseMapMenuState(MenuState)

Tila, jossa näytetään jokin karttavalikko. Alustus lisää kuvallisia valikkovaihtoehtoja karttojen lukumäärän mukaan ja lisäksi vaihtoehdon palata päävalikkoon.

Metodit:

start_game_with_map(map_index)
Luo uuden GameState -pelitilaolion ja siirtyy siihen.

GameOverMenuState(MenuState)

Tila, jossa näytetään pelin voittaja, kun peli loppuu. Alustus piirtää tekstin ja soittaa musiikin sen perusteella, kuka voitti pelin. Samoin lisätään vaihtoehto palata päävalikkoon.

Metodit:

draw()
Piirtää taustakuvan ja tekstit.

GameState(State)

Varsinainen pelitila. Alustus luo Map-olion eli kartan asetustenlukijan avulla, tutkii pelataanko tekoälyä vastaan, luo tapahtumankäsittelijän pelille, ja käynnistää pelin.

Metodit:

check_for_win()
Tarkastaa, onko molemmissa joukkueissa hahmoja elossa. Jos ei, siirtää ohjelman tilaan, jossa näytetään pelin voittaja.

update()
Kutsuu edellistä metodia pelin loppumisen tarkistamiseksi, ja kutsuu pelattavan Map-olion MapView-olion update()-metodia pelin elementtien päivittämiseksi.

draw()
Piirtää taustavärin, ja kutsuu pelattavan Map-olion MapView-olion draw()-metodia kartan piirtämiseksi. Sen lisäksi piirtää mahdolliset peliin liittyvät tekstit näytölle.

EventHandler

Toimii yliluokkana tapahtumankäsittelijöille. Niiden avulla käsitellään esimerkiksi hiiren ja näppäimistön painallukset.

Metodit:

handle(event)
Jakajametodi, joka vie edelleen tapahtumapinosta sille annetun tapahtuman oikealle käsittelijämetodille riippuen siitä onko tapahtuma näppäimistön painallus, hiiren painallus vai muu.

IntroEventHandler(EventHandler)

Tapahtumankäsittelijä ”intro screen” ja ”credits” -tiloille. Kaikki hiiren ja näppäimistön painallukset vievät päävalikkoon.

Metodit:

handle_click(mouse_pos)
Hiiren klikkaus vie päävalikkoon.

handle_key(key)
Näppäimistön painallus vie päävalikkoon.

MenuEventHandler(EventHandler)

Tapahtumankäsittelijä valikoille. Tunnistaa hiiren klikkaukset valikkovaihtoehtojen päällä ja myös hiiren kulkemisen vaihtoehdon päälle, jolloin se korostetaan.

Metodit:

handle_click(mouse_pos)
Tunnistaa, mitä valikkovaihtoehtoa klikattiin, ja suorittaa siihen liittyvän funktion, jos vaihtoehtoa ei ole poistettu käytöstä.

handle_event(event)
Tunnistaa hiiren liikkeet valikkovaihtoehtojen päälle ja korostaa vaihtoehdon, jonka päälle hiiri liikkuu.

GameEventHandler(EventHandler)

Tapahtumankäsittelijä itse pelille. Tunnistaa tapahtumat, joita pelaaja tai tekoäly tekee ja ohjaa peliä sen mukaan.

Metodit:

handle_event(event)
Tunnistaa tekoälyn tapahtumapinoon lisäämät tapahtumat, joilla hallitaan tekoälyhahmojen liikkeitä ja toimintoja.

handle_key(key)
Tunnistaa Esc-näppäimen painalluksen pelissä, asettaa pelin taukotilaan ja siirtyy päävalikkoon. Tuo myös ”Resume Game” vaihtoehdon näkyviin.

handle_click(mouse_pos)
Tärkein pelin hallintaan käytetty metodi, joka tunnistaa hiiren klikkaukset pelin aikana. Tunnistaa, mitä pelaaja klikkaa, ja päivittää pelin tilaa sen perusteella, liikuttaa hahmoja, käyttää toimintoja ja käyttöliittymän painikkeita.

Menu

Valikko, joka sisältää valikkovaihtoehdot.

Metodit:

Tällä luokalla ei ole omia metodeita.

MenuOption

Valikkovaihtoehto, jolla on teksti ja johon on liitetty jokin funktio ja mahdollisesti parametri funktiolle. Alustuksessa rakennetaan annetusta tekstistä Pygamen Surface -pinnat, joita voidaan käyttää suoraan piirtämiseen näytölle.

Metodit:

set_images()
Luo annetusta tekstistä kolme pintaa: 1) normaalilla valikon tekstin värillä 2) värillä, joka näytetään hiiren ollessa vaihtoehdon päällä 3) värillä, jota käytetään, kun vaihtoehto ei ole käytettävissä. Värit on määritelty vakioina ”constants” -moduulissa.

set_rect()
Asettaa valikkovaihtoehdon paikan näytöllä sen mukaan, kuinka monta niitä on koko valikossa.

update()
Vaihtaa piirrettävää kuvaa sen mukaan, onko vaihtoehto käytettävissä ja onko hiiri sen päällä vai ei.

draw()
Piirtää valikkovaihtoehdon näytölle.

MenuOptionMapImage(MenuOption)

Valikkovaihtoehto, joka näyttää kuvan kartasta tekstin yläpuolella. Alustuksessa tutkitaan karttojen lukumäärä, lasketaan sen perusteella piirrettävän kuvan koko, ja piirretään itse kuva kartasta. Perii tavalliselta MenuOption -luokalta, joten asetetaan ja piirretään myös tekstipinnat.

Metodit:

set_rect()
Asettaa valikkovaihtoehdon paikan näytöllä sen mukaan, kuinka monta karttaa on saatavilla.

draw()
Piirtää valikkovaihtoehdon kuvan kanssa näytölle.

Map

Kartta, jonka päälle peli rakentuu. Alustuksessa luodaan pelille vuoronhallitsija eli TurnController -luokan olio. Kartta sisältää ruudut, objektit sekä hahmot.

Metodit:

build_squares(height, width, squares_input, team1_start, team2_start, init_view=True)
Rakentaa kartan annettujen tietojen perusteella, eli kasaa ruudut ja niiden päälle mahdolliset objektit. Alustaa myös piirtämisestä huolehtivan MapView-olion, jos ei toisin haluta.

start_game()
Päivittää ensimmäisen kerran pelin elementeistä liikkumisen kantaman ilmaisimet, toimintonapit sekä hahmojen tiedot. Laukaisee myös ”PLAY!” -tekstin näytölle.

square_at(coordinates)
Palauttaa ruudun, joka on annetuissa koordinaateissa, jos mahdollista.

add_squaretype(squaretype)
Lisää kartan käytettäväksi uuden ruututyypin.

add_object_type(object_type)
Lisää kartan käytettäväksi uuden esinetyypin.

add_character (character, coordinates, facing, init_view=True)
Lisää kartalle hahmon annetuilla tiedoilla, jos mahdollista. Alustaa myös hahmolle piirtämisestä huolehtivan CharacterView-olion, jos ei toisin haluta.

add_object(coordinates, map_object)
Lisää kartalle annetun objektin annettuun paikkaan, jos mahdollista.

contains_coordinates(coordinates)
Kertoo, löytyykö kartalta annetut koordinaatit.

set_in_range(max_range, start, ignore_characters=False, ignore_non_walkable=False)
Asettaa karttaolion ”in_range” -attribuutin sisältämään kaikki koordinaatit, jotka ovat annetun maksimietäisyyden sisällä lähtöpisteestä. Halutessaan voi asettaa algoritmin olemaan välittämättä hahmoista tai ruututyypeistä, joissa ei voi kävellä. Algoritmin toiminta on kuvattu seuraavassa luvussa tarkemmin.

get_shortest_path(start, end, ignore_range=False)
Palauttaa lyhimmän reitin annettujen koordinaattien välillä. Olettaa, että edellinen metodi on ajettu ennen tätä, jotta ruutujen etäisyysluvut (”range_count” -attribuutti) ovat oikein, ja näin nopeutetaan algoritmien yhteistoimintaa. Voitaisiin myös sisällyttää tähän metodiin kutsu edelliseen metodiin aina alussa, mutta tässä pelissä oletus ei aiheuta ongelmia, koska ne ajettaisiin muutenkin aina peräkkäin, joten vältetään siis mieluummin turhaa toistoa.

get_simple_map()
Palauttaa kartan merkkijonoesityksenä niin, että jokaista ruutua kohden yksi kirjain kuvaa ruututyyppiä, objektia tai hahmoa, ja jokainen rivi ruutuja on omalla rivillään.

get_range_count_map()
Palauttaa edellistä vastaavan merkkijonoesityksen kartasta sillä erotuksella, että ruudun kohdalle tulostetaan sen ”range_count” -attribuutin arvo. Erityisesti algoritmien testaukseen tarkoitettu.

MapView

Tämä luokka vastaa kartan piirtämiseen liittyvistä toiminnoista ja on päävastuussa itse pelin käyttöliittymän piirtämisestä. Alustuksessa liitetään näkymä karttaolioon, määritetään joitakin tarpeellisia mittoja, ja ladataan resursseja, joita käytetään käyttöliittymässä.

Metodit:

center_in_window()
Keskittää kartan näytöllä.

load_char_info_backgrounds()
Lataa pelin ylälaidassa hahmojen tiedojen taustakuvina käytetyt kuvatiedostot.

load_indicators()
Lataa kantaman sisällä olevien ruutujen näyttämiseen käytetyt kuvatiedostot.

load_bottom_menu_bg()
Lataa pelin alalaidan valikon taustakuvat kuvatiedostoista ja piirtää ne ensimmäisen kerran.

draw_squares()
Piirtää kartan ruudut sitä varten määritellylle pinnalle, jotta myöhemmin voidaan vain piirtää tämä pinta eikä kaikkia ruutuja yksitellen.

draw_range()
Piirtää kantaman sisällä olevien ruutujen ilmaisimet sitä varten määritellylle pinnalle, jotta voidaan vain piirtää tämä pinta eikä kaikkia ilmaisimia yksitellen.

update_action_buttons()
Päivittää vasemmassa alanurkassa olevat toimintojen käyttämiseen tarkoitetut napit vastaamaan vuorossa olevan hahmon toimintoja.

update_char_info()
Päivittää ylälaidan hahmojen tiedot vastaamaan uutta tilannetta.

trigger_effect_text(text, color, coordinates)
Luo tekstipinnan, joka ilmaisee käytetyn toiminnon vaikutusta kohdehahmoon. Teksti piirretään kohdekoordinaattien päälle.

update_effect_text()
Päivittää edellisen metodin luomaa tekstipintaa niin, että teksti leijuu ylöspäin ja katoaa hetken kuluttua.

trigger_event_text()
Luo tekstipinnan, joka näytetään suurena tekstinä pelin päällä esimerkiksi vuoron vaihtuessa pelaajan hahmolle.

update_event_text()
Näyttää edellisellä metodilla laukaistun tekstin 1.5 sekunnin ajan, jonka jälkeen se katoaa.

update()
Kutsuu kaikkia tarvittavia päivitysmetodeita, kun tätä kutsutaan pelin näkymän päivittämiseksi. Kutsuu siis yllämainittuja update-metodeita ja sen lisäksi hahmojen ja nappien update-metodeita.

draw()
Hoitaa pelin näkymän piirtämisen näytölle piirtämällä elementtejä itse ja kutsumalla hahmojen, objektien ja nappien omia draw-metodeita.

Button

Määrittää pelin käyttöliittymässä käytettävän napin. Alustuksessa ladataan kuvat kaikille hiiren tiloille (normaali, hiiri napin päällä, painallus) ja ladataan napin teksti. Polut kuvatiedostoihin ja tekstin tiedot annetaan parametreina luotaessa oliota.

Metodit:

set_rect(pos)
Sijoittaa napin annettuun paikkaan näytöllä.

update()
Valitsee piirrettäväksi napin kuvan hiiren tilan mukaan, korostaa napin hiiren ollessa sen päällä ja painaa napin pohjaan kun hiirtä painetaan.

draw()
Piirtää napin näytölle.

SquareType

Määrittää ruututyypin, jota voidaan käyttää kartassa. Alustuksessa asetetaan nimi, lyhenne, voiko ruututyypin päällä kävellä, ja ladataan kuva.

Metodit:

__eq__(obj)
Korvaa normaalin vertailumetodin, jotta saman ruututyypin mahdolliset eri oliot tunnistetaan samaksi ruututyypiksi luokan ja nimen perusteella.

__str__()
Palauttaa ruutytyypin nimen.

Square

Määrittää karttaruudun. Alustuksessa määritetään ruudun koordinaatit ja tyyppi.

Metodit:

is_empty()
Palauttaa True tai False riippuen siitä, onko ruutu tyhjä, eli onko ruudussa hahmoa tai objektia.

__str__()
Palauttaa ruudun ruututyypin nimen.

ObjectType

Määrittää objektityypin, jota voidaan käyttää kartalla. Alustuksessa määritetään tyypin nimi, lyhenne, kuva, ja piirtämisessä käytettävät siirrokset. Siirroksilla (offset_x ja offset_y) mahdollistetaan erikokoisten objektien luominen niin, että voidaan erikseen määrittää mihin kohtaan ruutua ne tulisi piirtää.

Metodit:

__str__()
Palauttaa objektityypin nimen.

MapObject

Määrittää halutun tyyppisen karttaobjektin. Alustettaessa annetaan vain objektin tyyppi.

Metodit:

added_to_map(map, coordinates)
Päivittää karttaobjektin attribuutteihin kartan ja koordinaatit, joille se lisättiin.

set_view()
Alustaa objektille näkymän, MapObjectView-luokan olion, joka vastaa piirtämisestä.

__str__()
Palauttaa objektin tyypin nimen.

MapObjectView

Määrittää karttaobjektin näkymän, joka vastaa piirtämisestä kartalle. Alustettaessa liitetään näkymä objektiin, ladataan objektityypin kuva, ja asetetaan sijoittelu näytöllä.

Metodit:

set_screen_pos()
Asettaa objektin piirrettäväksi oikeaan kohtaan kartalle sen koordinaattien perusteella.

draw()
Piirtää objektin näytölle.

Character

Määrittää pelin hahmon. Hahmolle annetaan alustettaessa nimi, joukkue, ”maksimiterveys”, liikkumisen kantama, mahdollinen tekoäly sekä polut kuvatiedostoihin, joista se tulisi piirtää.

Metodit:

added_to_map(map, coordinates, facing, init_view=True)
Päivittää hahmo-olion attribuutit, kun se lisätään kartalle. Liittää hahmon karttaan, asettaa koordinaatit ja suunnan johon se on kääntyneenä. Asettaa myös hahmon näkymän, jos ei toisin haluta.

set_view()
Alustaa hahmolle näkymän, CharacterView -luokan olion.

damage(amount)
Vähentää hahmon ”elämää” annetulla luvulla, ja tarkastaa kuoliko hahmo.

stun(turns)
Tainnuttaa hahmon annetuksi määräksi vuoroja.

heal(amount)
Lisää hahmon ”elämää” annetulla luvulla, ylärajana hahmolle annettu maksimiterveys.

buff(multiplier)
Asettaa hahmon toimintojen vahvuuden kertoimen annetuksi luvuksi. Toisin sanoen vahvistaa tai heikentää toimintojen vaikutusta.

add_action(action_type, strength, action_range, name, sound=None)
Lisää hahmolle toiminnon annetuilla tiedoilla. Ääniefekti on vapaaehtoinen.

turn_to_direction(direction)
Kääntää hahmon annettuun suuntaan.

turn_towards(coordinates)
Kääntää hahmon kohti annettuja koordinaatteja, vaikka ne eivät olisi suoraan hahmon vieressä.

end_turn()
Antaa vuoron seuraavalle hahmolle ja laukaisee vuoron vaihtumisesta johtuvat käyttöliittymän elementtien päivitykset.

move_forward()
Liikuttaa hahmoa kartalla yhden ruudun suuntaan, johon se on kääntyneenä, jos liikkuminen siihen suuntaan on mahdollista.

move_to_direction(direction)
Liikuttaa hahmoa kartalla yhden ruudun annettuun suuntaan, jos liikkuminen siihen suuntaan on mahdollista.

move_to_coordinates(target)
Liikuttaa hahmon annettuihin kohdekoordinaatteihin, jos ne ovat liikkumisen kantaman sisällä. Käytetään yksikkötestaamisessa apuna.

__str__()
Palauttaa hahmon nimen.

CharacterView

Määrittää hahmon näkymän, joka huolehtii niiden piirtämisestä kartalle. Alustettaessa liittää näkymän hahmoon, lataa kaikki piirtämisessä käytetyt kuvat valmiiksi, asettaa ensimmäisen kerran sijoittelun näytöllä, ja luo animaation apuna käytettäviä muuttujia.

Metodit:

set_screen_pos()
Asettaa paikan näytöllä niin, että se vastaa hahmon koordinaatteja kartalla.

get_next_walk_frame()
Valitsee ja palauttaa seuraavan kävelyanimaation kuvan.

update()
Päivittää hahmon ruudulle piirrettävää kuvaa. Tunnistaa, kun hahmo kävelee, ja päivittää animaation mukana kuvaa sekä hahmon paikkaa kartalla myös taustalogiikassa. Jos kyseessä on tekoälyhahmo, vie myös kävelyn päätteeksi tapahtumapinoon tapahtuman, joka laukaisee tekoälyn toiminnon käyttämisen.

draw()
Piirtää hahmon kuvan ruudulle.

Action

Määrittää toiminnon, joka vaikuttaa toisiin hahmoihin. Luokkaan on määritelty neljä toimintotyyppiä: damage, heal, stun, ja buff. Nämä vaikuttavat hahmoihin Character-luokan kuvauksessa kerrotulla tavalla. Toiminnolle asetetaan alustettaessa hahmo, jolle se kuuluu, tyyppi, voimakkuus, kantama, nimi sekä mahdollinen ääniefekti.

Metodit:

perform(target_coordinates)
Suorittaa toiminnon annetuissa kohdekoordinaateissa. Kääntää hahmon suorituksen suuntaan, laukaisee näytölle voimakkuudesta kertovan tekstin, joka leijuu ylöspäin, ja soittaa mahdollisen ääniefektin. Nollaa myös ”buff” -toiminnon muuttaman hahmon toimintojen kertoimen takaisin yhteen.

__str__()
Palauttaa toiminnon nimen.

TurnController

Hallitsee pelin vuoroja ja on myös päävastuussa tekoälyhahmojen toiminnan kontrolloimisesta. Alustettaessa TurnController-olio liitetään karttaan, luodaan apumuuttujia ja alustetaan Ai-luokan olio tekoälyä varten.

Metodit:

add_character(character)
Lisää vuoronhallintaan annetun hahmon.

reset()
Nollaa vuoronhallinnan ja antaa vuoron ensimmäisenä lisätylle hahmolle.

next()
Etenee pelissä seuraavaan vuoroon. Päivittää pelitilan, jolloin tarkastetaan voittiko toinen joukkue, ja jos peli ei loppunut eli kumpikaan joukkue ei voittanut vielä, antaa vuoron seuraavalle hahmolle. Jos seuraava hahmo on tainnutettu tai kuollut, hypätään sen vuoron yli. Jos seuraava hahmo taas on tekoälyhahmo, lisätään Pygamen tapahtumapinoon tapahtuma, joka laukaisee tekoälyhahmon liikkumisen. Jos vuoro tuli pelaajan hahmolle, näyttää tekstin, joka ohjeistaa liikkumaan tai valitsemaan toiminnon.

ai_move()
Hakee Ai-luokan olion avulla tekoälyhahmolle liikuttavan polun. Jos polku saatiin, laukaistaan hahmon kävely. Jos ei saatu polkua, asetetaan Pygamen tapahtumapinoon tapahtuma, joka laukaisee tekoälyhahmon toiminnon käyttämisen.

ai_use_action()
Hakee Ai-luokan olion avulla toiminnon ja kohteen. Jos toiminto saatiin, se suoritetaan kohteessa ja lopetetaan vuoro. Jos ei saada toimintoa, lopetetaan vain vuoro.

Ai

Tässä luokassa määritetään tekoälyn algoritmit, joiden avulla haetaan kohdehahmot ja käytettävät toiminnot.

Metodit:

get_target(get_team_member=False)
Hakee toisesta joukkueesta kohdehahmon seuraavassa luvussa kuvatun algoritmin avulla. Parametrina voidaan asettaa, että halutaan kohde omasta joukkueesta. Palauttaa hahmon ja polun siihen.

get_next_move()
Hakee ensin kohdehahmon ja polun siihen edellisen metodin avulla, ja ottaa saadusta polusta sitten askeleet, jotka ovat kantaman sisällä, jolloin kuljetaan polkua niin pitkälle kuin yhden vuoron aikana päästään.

choose_action(team_action=False)
Ensimmäinen osa tekoälyn käyttämän toiminnon hakemista, toimii apumetodina seuraavalle metodille ”get_action()”. Valitsee toiminnon hahmolle määritellyistä toiminnoista seuraavassa luvussa esitellyn algoritmin avulla. Parametrina voidaan asettaa, että halutaan toiminto, joka auttaa omaa joukuetta. Palauttaa toiminnon ja kohdehahmon.

get_action()
Hakee edellisen metodin avulla ensin sekä hyökkäystoiminnon että omaa joukkuetta auttavan toiminnon, ja valitsee sitten näistä tekoälyhahmolle suoritettavan toiminnon seuraavassa luvussa tarkemmin kuvatulla tavalla. Palauttaa valitun toiminnon ja kohdehahmon.

Coordinates

Määrittää karttakoordinaattiparin ja hyödyllisiä apumetodeita. Attribuutteina määritellään x- ja y-koordinaatti.

Metodit:

get_neighbor(direction)
Hakee annetussa suunnassa olevat naapurikoordinaatit ja palauttaa näistä koordinaattiolion.

get_neighbors()
Hakee kaikissa neljässä suunnassa olevat naapurikoordinaatit ja palauttaa koordinaatioliot.

is_neighbor_at(coordinates, direction)
Kertoo, onko annettu koordinaattiolio tämän olion naapuri tietyssä suunnassa.

ConfigFileReader

Määrittää asetustiedostonlukijan, jolla luetaan kartta- ja hahmo-asetustiedostot ja rakennetaan näiden perusteella kartat ja hahmot.

Metodit:

check_parentheses(input)
Tarkistaa annetusta syötteestä erilaisten sulkujen oikean käytön, ja ilmoittaa virheen, jos syötteessä on parittomia sulkuja.

read_config()
Lukee syötteen ja muodostaa sen osioista listan. Jokaisesta osiosta muodostetaan sanakirja, johon sen asetukset ja arvot talletetaan. Tämän metodin toiminta selostetaan tarkemmin luvussa 7, Tiedostot.

build_map_base(state, map_config, map_index)
Rakentaa annetusta syötteestä karttapohjan, eli kartan ilman hahmoja. Lisää ensin karttaan ruututyypit ja objektityypit, ja sen jälkeen rakentaa Map-luokan avulla kartan. Palauttaa rakennetun karttaolion. Avustaa get_map() -metodia.

build_characters_on_map(character_config, map)
Rakentaa annetusta syötteestä hahmot toimintoineen ja lisää ne annetulle kartalle. Avustaa get_map() -metodia.

count_available_maps()
Laskee kuinka monta karttaa on saatavilla ja palauttaa lukumäärän.

get_map(state, map_index)
Rakentaa luettujen asetustiedostojen perusteella halutun kartan ja palauttaa sen. Käyttää hyväkseen edellä kuvattuja build_map_base() ja build_characters_on_ma() -metodeita.

ConfigFileError

Määrittää tiedostonluvun virheistä nostettavan poikkeuksen.

Metodit:

Tällä luokalla on ainoastaa alustusmetodi, joka kutsuu Exception -yliluokkaa parametrina annetun viestin kanssa.

5. Algoritmit

Peli käyttää muutamaa olennaista algoritmia erityisesti etäisyyksien ja polkujen etsintään sekä tekoälyyn.

Etäisyyden sisällä olevien ruutujen etsiminen, metodi Map.set_in_range()

Map-luokassa oleva metodi set_in_range() etsii ne ruudut, jotka ovat annetun etäisyyden sisällä annetusta lähtöpisteestä. Parametreina voi myös halutessaan asettaa, että mukaan luetaan ruudut, joihin ei voi kävellä tai ruudut, joissa on jokin hahmo. Oletuksena tällaisia ruutuja ei oteta mukaan.
Metodi tekee toimintansa aikana kaksi hyödyllistä asiaa. 1) Se asettaa kyseisen Map-olion attribuutin ”in_range” viittaamaan listaan niiden ruutujen koordinaateista, jotka ovat annetun etäisyyden sisällä. 2) Tämän sivutuotteena se asettaa jokaisen ruudun attribuutin ”range_count” etäisyydeksi lähtöpisteestä. Tätä ”range_count” attribuuttia voidaan hyödyntää hyvin lyhimmän polun etsivässä metodissa.

Käytettävä algoritmi pohjautuu läheisesti BFS-hakualgoritmiin. Käydään leveyssuuntaisesti läpi vierekkäisiä ruutuja, ja lisätään niitä palautettavaan listaan, jos ne täyttävät halutut ehdot. Tässä tapauksessa nämä ehdot liittyvät siihen, voiko ruudussa kävellä tai onko siinä hahmo tai objekti. Vierellä olevat ruudut lisätään aina jonoon, ja valitaan aina jonon vanhin ruutu seuraavaksi, jolloin käytännössä käydään ruutuja läpi kerroksittain aina tietyn etäisyyden päässä olevat ruudut kerrallaan. Kun ruudussa vieraillaan, sen vierellä olevat ruudut vastaavasti lisätään jonoon, jos ne täyttävät ehdot ja jos niissä ei ole jo vierailtu. Jos tähän vaiheeseen lisätään vielä ehto, joka seuraa etäisyyttä lähtöpisteeseen, voidaan estää sellaisten ruutujen läpikäynti, jotka ovat liian kaukana. Jokaiseen uuteen ruutuun saapuessa asetetaan ruutuolion ”range_count” attribuutti yhdeksi enemmän kuin edellisessä ruudussa, ja verrataan tätä luku annettuun maksimietäisyyteen. Näin saadaan kerättyä sellaiset ruudut, jotka ovat riittävän lähellä esteet huomioiden.

Metodin parametreissa voi asettaa halutessaan, että mukaan otetaan myös ruudut, jotka ovat tyyppiä jossa ei voi kävellä tai ruudut, joissa on hahmo. Näin mahdollistetaan saman metodin käyttäminen sekä liikkumiselle että hahmojen käyttämille toiminnoille. Itse käytän tätä esimerkiksi niin, että toimintoja käytettäessä hahmo voi ampua myös ruututyyppien yli joissa se ei voi kävellä, jolloin voidaan asettaa parametreihin ehto, että otetaan mukaan myös ruututyypit, joissa ei voi kävellä.

Karttaolion ”in_range” attribuuttia kysytään melko usein, joten tämä metodi asettaa tuloksensa suoraan siihen talteen, jolloin sitä ei tarvitse kutsua joka kerta uudelleen, kun lista koordinaateista halutaan tietää.

Tätä metodia hyödynnetään esimerkiksi, kun näytetään ruudut, joihin hahmo voi liikkua, ja ruudut, joihin toimintoa voi käyttää.

Lyhimmän polun etsiminen, metodi Map.get_shortest_path()

Map-luokassa oleva metodi get_shortest_path() etsii lyhymmän polun kartalla annetusta lähtöpisteestä annettuun loppupisteeseen. Tämä metodi käyttää hyödykseen Map.set_in_range() -metodin muuttamaa ruutuolioiden ”range_count” attribuuttia.

Polun etsimiseen käytettävä algoritmi pohjautuu Lee algoritmiin, joka on hyvin tunnettu perusalgoritmi lyhimmän reitin etsimiseen. Tässä voitaisiin käyttää myös edistyneempiä algoritmeja, kuten Dijkstran algoritmia tai A* algoritmia, mutta pelin ruudukko on melko pieni, joten Lee algoritmi on hyvin riittävä tähän tapaukseen ja yksinkertainen ymmärtää.

Lähdetään siis loppupisteestä liikkeelle, ja jokaisessa ruudussa käydään läpi vierellä olevia ruutuja ja siirrytään aina sellaiseen viereiseen ruutuun, jonka etäisyys alkupisteestä on pienempi kuin nykyisen ruudun. Näin siirrytään aina askel kerrallaan lähemmäksi alkupistettä, ja jatketaan, kunnes jokin viereinen ruutu on etsitty alkupiste. Saadaan kerättyä lyhin reitti loppupisteestä alkupisteeseen, ja kun tämä käännetään toisin päin, saadaan lyhin reitti alkupisteestä loppupisteeseen.

Tätä metodia hyödynnetään hahmon liikkumisessa sekä tekoälyn kohteen etsimisessä. Hahmoa liikuttaessa etsitään ensin mahdolliset ruudut, ja kun ruutu valitaan, tämä metodi etsii lyhimmän reitin siihen, jotta hahmo osaa varsinaisesti sitten liikkua siihen askel kerrallaan löydettyä polkua pitkin.

Tekoälyn kohteen etsintä, metodi Ai.get_target()

Tekoäly etsii vuorossa olevalle hahmolle kohteen hahmojen sijainnin ja hahmojen kunnon perusteella niin, että lähempänä olevat ja heikommat hahmot tulevat valituksi helpommin. Koska etäisyyksiä ja hahmojen jäljellä olevaa ”elämää” ei voida suoraan verrata, annetaan mahdollisille kohteille ranking näiden ominaisuuksien perusteella ja kahdesta rankingista otetaan keskiarvo. Kohteeksi valitaan hahmo, jolla on pienin keskiarvoranking. Tasatilanteessa valitaan lähempi hahmo. Algoritmi toimii seuraavalla tavalla.

Valitaan kohdetta etsiväksi hahmoksi nyt vuorossa oleva hahmo. Käydään sitten läpi kaikki kartalla olevat hahmot, ja valitaan mahdolliset kohteet. Liikkumista ja hyökkäyksiä varten tässä valitaan toisessa joukkueessa ja elossa olevat hahmot, mutta metodin parametreissa voi määrittää haluavansa oman joukkueen hahmot, mitä käytetään parantavien ja vahvistavien toimintojen kanssa.

Kun mahdolliset kohdehahmot on löydetty, tarkastetaan että ne ovat saavutettavissa: asetetaan kartan kaikille ruuduille Map.set_in_range() -metodin avulla etäisyysluvut ja tarkastetaan, että ruutu, jossa kukin mahdollinen kohde on, on saanut etäisyysluvun, joka eroaa nollasta. Tässä ei oteta huomioon rajoituksia hahmojen askelmäärissä yhden vuoron aikana, vaan haetaan koko kartan kattavuus, jotta saadaan huomioitua kaikki hahmot, joiden luo kohdetta etsivän hahmon olisi mahdollista liikkua edes usean vuoron aikana. Jos hahmo on saavutettavissa, haetaan etäisyys siihen Map.get_shortest_path() -metodin avulla ja otetaan sekä saadun polun pituus että hahmon jäljellä oleva ”elämä” talteen seuraavia vaiheita varten. Jos taas hahmo ei ole saavutettavissa, poistetaan se mahdollisista kohteista. Seuraavaksi tehdään tarkistus, että yhtään mahdollisia kohteita on jäljellä.

Saavutettavissa olevat mahdolliset kohteet järjestetään seuraavaksi erikseen etäisyyden ja ”elämän” perusteella. Järjestämisen jälkeen annetaan hahmoille ranking niin, että sama etäisyys tai ”elämä”, eli tasatilanne, johtaa samaan rankingiin. Rankingeista lasketaan sitten keskiarvo, ja kohteeksi valitaan se hahmo, jolla on pienin keskiarvoranking. Tasatilanteessa valitaan lähempi hahmo. Lopuksi haetaan vielä lyhin polku valittuun kohteeseen ja palautetaan sekä hahmo että polku. Alla on vielä kerrattuna algoritmin vaiheet.

1) valitaan mahdolliset kohteet
2) haetaan etäisyydet hahmoihin ja niiden jäljellä oleva elämä
3) järjestetään saavutettavissa olevat hahmot etäisyyden ja elämän perusteella
4) annetaan ranking etäisyyksien ja elämän perusteella
5) otetaan rankingeista keskiarvo
6) valitaan kohteeksi se, jolla on pienin keskiarvo, tasatilanteessa lähempi

Tekoälyn käytettävän toiminnon valinta, metodi Ai.get_action()

Tekoäly valitsee vuorossa olevalle hahmolle käytettävän toiminnon sillä periaatteella, että käytetään mahdollisimman tehokasta toimintoa. Toisin sanoen valitaan sellainen toiminto, joka on mahdollisimman tehokas niistä toiminnoista, jotka ovat mahdollisia ottaen huomioon toisten hahmojen sijainnin kartalla. Jos esimerkiksi viereisessä ruudussa on vastustajan hahmo, kaikki hyökkäykset ovat yleensä mahdollisia, mutta vastustajan hahmo saattaa olla myös kauempana, jolloin joudutaan käyttämään heikompaa hyökkäystä, jolla on pidempi kantama. Valinnassa on mukana myös suhteellisen pieni satunnaiselementti, joka johtaa siihen, että tekoäly on suurimman osan ajasta rationaalinen, mutta tekee joskus myös epärationaalisia ratkaisuja. Se osaa myös ottaa huomioon mahdolliset joukkuetovereihin kohdistuvat toiminnot. Jos olisi mahdollista sekä hyökätä että tehdä jokin omaan joukkueeseen kohdistuva toiminto, tekoäly on noin neljänä viidestä kerrasta aggressiivinen ja yhtenä viidestä kerrasta auttaa joukkuetovereita. Toiminnon valinta tapahtuu seuraavalla tavalla.

Päämetodi Ai.get_action() kutsuu ensin apumetodia Ai.choose_action(), joka tekee valinnan mahdollisista hyökkäyksistä ja joukkuetoiminnoista. Tätä siis kutsutaan kahdesti, kerran haetaan hyökkäystä ja kerran joukkutoimintoa. Jos molemmat palauttavat jonkin toiminnon, valitsee tekoäly 80% ajasta hyökkäyksen ja muuten joukkuetta auttavan toiminnon. Jos saadaan vain hyökkäys tai joukkuetoiminto, valitaan se. Ja luonnollisesti, jos ei saada kumpaakaan toimintoa, ei mitään toimintoa voida käyttää. Itse apumetodi toimii seuraavalla tavalla. Sen parametrina voidaan asettaa, jos halutaan joukkuetoiminto hyökkäyksen sijasta.

Valitaan ensin toiminnolle kohde edellä kuvatun Ai.get_target() -metodin avulla, hyökkäystä varten vastustajan hahmo ja joukkuetoimintoa varten joukkuetoveri. Käydään sitten kaikki hahmon hyökkäykset tai joukkuetoiminnot läpi ja valitaan niistä se, jolla on pisin kantama. Jos ei löydetä toimintoa, lopetetaan tähän. Muuten tämän kantaman avulla käytetään Map.set_in_range() -metodia, joka alustaa kartan ruudut niin, että saadaan seuraavaksi haettua lyhin polku kohteeseen. Jos polkua ei saatu, tarkoittaa se, että kohde on liian kaukana, ja voidaan lopettaa tähän. Muuten otetaan saadun polun pituus talteen ja jatketaan valintaa.

Käydään seuraavaksi kaikki halutunlaiset toiminnot läpi, ja valitaan niistä se, jolla on suurin vaikutus ja silti tarpeeksi kantamaa verrattuna etäisyydeen kohdehahmoon. Tässä annetaan oletuksia vaikutusten suuruudesta, jotta erityyppisen toiminnot olisivat verrattavissa. Kaikkien toimintotyyppien oletukset on lueteltu alla.

Joukkuetoiminnot:

Heal:
Toiminnon oma voimakkuus

Vahvistava Buff:
10 x Toiminnon oma voimakkuus

Hyökkäykset

Damage:
Toiminnon oma voimakkuus

Stun:
Voimakkuus 20

Heikentävä Buff:
10 / Toiminnon oma voimakkuus

Kun kaikki hyökkäykset tai joukkuetoiminnot on käyty läpi, on saatu valituksi toiminto, jolla on suurin vaikutus, ja palautetaan se ja valittu kohde.

Tekoäly osaa siis mukautua hahmoille annettujen toimintojen mukaan, ja tehdä suurimman osan ajasta rationaalisia päätöksiä, joskus kuitenkin tehden huonompia valintoja. Oletusten avulla saadaan skaalattua toimintotyyppien vahvuuksia vertailukelpoisiksi. Tekoäly on siinä mielessä aggressiivinen, että se liikkuu ensin joka tapauksessa vastustajan hahmoa kohti, ja sen jälkeen valitsee toiminnon. Toisin sanoen se ei lähde liikkumaan omia hahmoja kohti esimerkiksi parantaakseen niitä, vaan saattaa käyttää joukkuetoimintoja vain silloin, kun oma hahmo on lähellä esimerkiksi taistelemassa saman vastustajan hahmon kanssa.

Tässä vielä kerrattuna vaiheet, apumetodin toiminta sisennettynä ensimmäisen vaiheen alla:

1) Haetaan sekä hyökkäys että joukkuetoiminto apumetodin avulla
    1. Haetaan kohdehahmo (jos ei saada hahmoa, lopetetaan)
    2. Haetaan toiminto, jolla on pisin kantama (jos ei toimintoja, lopetetaan)
    3. Tarkastetaan, että kohdehahmo on pisimmän kantaman sisällä
    4. Käydään kaikki toiminnot läpi ja valitaan se, jolla on suurin vaikutus, mutta silti riittävä kantama
2) Jos saatiin sekä hyökkäys että joukkuetoiminto, valitaan 80% ajasta hyökkäys. Jos saatiin vain toinen, valitaan se.
3) Jos valittiin yhtään toimintoa, palautetaan se ja kohdehahmo.

Isometriset muunnokset

Pelin grafiikat piirretään isometrisessä perspektiivissä, jolloin ohjelman aikana täytyy tehdä paljon muunnoksia näyttökoordinaateista karttakoordinaateiksi ja päinvastoin. Tähän käytetään kahta apumetodia ”coordinates” -moduulissa. Muunnokset tehdään pohjimmiltaan alla esitettyjen kaavojen perusteella. Ohjelman toiminnassa tehdään jotakin siirroksia johtuen esimerkiksi kartan sijoittelusta näytöllä ja siitä, että piirtotasolle (”surface”) voidaan piirtää vain positiivisiin koordinaatteihin, jos halutaan kaiken tulevan näkyviin. Nämä siirrokset riippuvat jatkuvasti tilanteesta, joten tässä on esitetty vain pohjalla olevat kaavat.

Kartalta näytölle (karteesinen -> isometrinen):

$$x_{näyttö} = (x_{kartta} - y_{kartta}) * (ruudun\ leveys/2)$$
$$y_{näyttö} = (x_{kartta} + y_{kartta}) * (ruudun\ korkeus/2)$$

palautetaan $x_{näyttö}$ ja $y_{näyttö}$

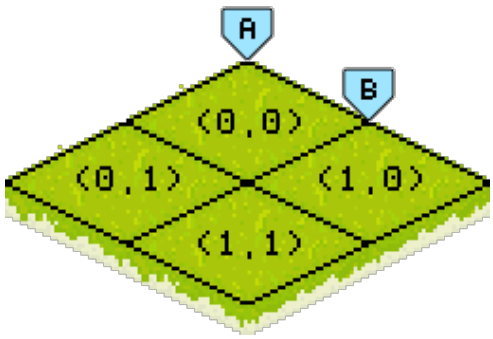
Näytöltä kartalle (isometrinen -> karteesinen):

$$x_{kartta} = \left(y_{näyttö} + x_{näyttö}/2\right)/ruudun\ korkeus$$
$$y_{kartta} = \left(y_{näyttö} - x_{näyttö}/2\right)/ruudun\ korkeus$$

jos x_{kartta} tai y_{kartta} on negatiivinen, vähennetään niistä yksi

palautetaan int(x_{kartta}) ja int(y_{kartta})

Ruudun leveys ja korkeus ovat yhden ruudun korkeus ja leveys pikseleissä näytöllä, tässä tapauksessa ne on määritelty vakioiksi korkeus = 32px ja leveys = 64px. Esimerkki kaavojen toiminnasta:



Pisteessä A sekä karteesiset että isometriset koordinaatit ovat (0, 0). Etsitään isometriset koordinaatit pisteelle B eli ruudulle (1, 0):

$$x = (1 - 0) * (64/2) = 32$$
$$y = (1 + 0) * (32/2) = 16$$

Nähdään, että B on koordinaateissa (32, 16). Tehdään myös muutos takaisin:

$$x = \frac{16+32/2}{32} = 1$$
$$y = \frac{16-32/2}{32} = 0$$

Eli saadaan myös, että piste B (32, 16) vastaa oikein ruutua (1, 0).

Koska ruutujen keskeltä voidaan myös klikata, pitää näytöltä kartalle -muunnoksessa pyöristää lopputulokset kokonaisluvuiksi. Tässä käytetään Pythonin int() -metodia. Siksi täytyy vähentää negatiivisista koordinaateista yksi, sillä muuten pyöristettäessä saataisiin, että muunnoksen tulokset välillä]-1 , 0[pyöristyisivät nollaan. Toisin sanoen kartan ulkopuolelta klikkaaminen tunnistettaisiin kartan sisäpuoleksi, jos klikkaaminen tapahtuisi tarpeeksi lähellä kartan ulkolaitaa.

6. Tietorakenteet

Suunnitelman mukaisesti suurin osa ohjelman tiedosta on tallennettuna olioiden attribuuteissa, esimerkiksi hahmojen ja kartan tila ja toimintojen ominaisuudet. Ohjelma käyttää runsaasti Pythonin listoja ja sanakirjoja esimerkiksi kartan tietojen tallentamisessa ja metodien toiminnassa. Seuraavassa käsitellään erityiskohtia, joissa on käytetty jotain toisenlaista tapaa käsitellä ja varastoida tarvittavia tietoja.

Pelin toiminta perustuu karttaan, joka on ruudukko, joten siinä käsitellään runsaasti koordinaatteja. Koordinaateille onkin oma luokka ”Coordinates”, jonka attribuutteina ovat x ja y. Koordinaatteja voitaisiin periaatteessa ilmaista myös esimerkiksi monikkoina (x, y), mutta luokan avulla saadaan tehtyä helposti joitakin hyödyllisiä apumetodeja, joiden avulla saadaan selville esimerkiksi koordinaattiolioiden vierekkäisyyteen liittyviä asioita, kuten onko olio jonkin toisen olion naapuri tietyssä suunnassa, tai vaikkapa kaikki jonkin pisteen naapurikoordinaatit yhdellä metodikutsulla. Voidaan myös kysyä x- ja y-koordinaatteja olion attribuuteissa esimerkkinä muodossa koordinaatit.x tai koordinaatit.y. Tämä helpottaa merkittävästi koodin kirjoittamista ja lukemista, ja auttaa välttämään sekaannuksia indeksien kanssa.

Ruudukkoon liittyen on myös määritelty neljä suuntaa muuttumattomiin monikkoihin. Nämä on määritelty vakioina, koska suunnat eivät muutu missään tapauksessa. Suunnan monikko on muotoa

```
UP = ((0, -1), ”up”, (2, -1))
```

ja se sisältää järjestyksessä tiedot x- ja y-askel ruudukolla, suunnan nimi, ja viimeisenä suunnan x- ja y-askel isometrisesti. Viimeistä hyödynnetään erityisesti hahmojen kävelyssä kartalla, kun niiden kuvia siirretään näytöllä ruudukon suuntien mukaisesti. Se, että suunnat on talletettu vakioihin näin, helpottaa myös huomattavasti koodin kirjoittamista ja lukemista, kun käytetään vain vakion nimeä. Samoin voidaan huoletta käyttää suuntavakioita myös paikoissa, joissa tiedon pitää olla eri muodossa loppukäyttöpaikassa, esimerkiksi voitaisiin asettaa, että hahmo liikkuu suuntaan UP, jolloin saadaan indeksillä nolla koordinaattien muutos ruudukolla, ja indeksillä kaksi muutos näytöllä, ilman tarvetta tehdä useassa paikassa samoja muunnoksia erikseen. Samaan moduuliin ”direction” on myös määritelty apumetodeita suuntiin liittyviä kyselyitä varten, kuten x:n ja y:n muutos erikseen johonkin suuntaan mentäessä, ja metodi, joka palauttaa kaikki suuntamonikot.

Ohjelmassa on myös melko runsaasti sellaisia tietoja, joita käytetään usein ja halutaan voitavan asettaa asetustiedostomaisesti yhdessä paikassa helposti. Tällaiset tiedot on asetettu vakioiksi moduulissa ”constants”. Se sisältää esimerkiksi yleisiä tietoja, kuten ikkunan koko, ruudunpäivitysnopeuden yläraja, karttaruutujen koko pikseleissä, värejä, fontit, ja runsaasti polkuja ohjelman käyttämiin resursseihin, kuten kuvatiedostoihin, äänitiedostoihin ja asetustiedostoihin. On esimerkiksi erittäin kätevää kirjoittaa koodissa BLACK tai WHITE sen sijaan, että joka kerta määrittää erikseen käsin rgb-arvot. Samoin voidaan määrittää polut käyttöliittymän yleisiin grafiikoihin, jotka eivät ole kartan elementtejä tai hahmoja, kuten napit ja valikkoelementtien taustat. Koodin lukemisen ja kirjoittamisen helpottamisen lisäksi tällä mahdollistetaan tietojen muuttaminen yhdessä paikassa ilman, että täytyy etsiä niitä yksitellen koodin seasta. Ohjelmointiteknisesti nämä ovat muuttumattomia vakioita, sillä niiden ei tarvitse muuttua ohjelman ollessa käynnissä.

Edellisessä luvussa kuvattu etäisyyden sisällä olevien ruutujen etsimiseen käytetty metodi käyttää toiminnassaan jonoa. Toteuksena käytetään Pythonin valmista jonoluokkaa queue. Sen

avulla luodaan jono, lisätään siihen ruutuja, ja otetaan niitä pois FIFO -periaatteella breadth first -ajatuksen mukaisesti.

7. Tiedostot

Ohjelma käsittelee neljänlaisia tiedostoja: kuvatiedostoja, äänitiedostoja, tekstimuotoisia asetustiedostoja ja lisäksi fonttitiedostoja (oletuksena vain yhtä).

Kuvatiedostoja käytetään pelin graafisen käyttöliittymän piirtämiseen. Kuvatiedostot ovat joko .gif tai .png -formaatissa. Ohjelma ei erityisesti vaadi näitä formaatteja, mutta niitä käytettäessä läpinäkyvyys otetaan huomioon. Oletusarvona kuvatiedostojen odotetaan olevan ”graphics” -kansiossa.

Äänitiedostoja käytetään musiikkien sekä ääniefektien toistamiseen. Äänitiedostot ovat .ogg -muodossa, jolla on mahdollisimman hyvä tuki useille käyttöjärjestelmille Pygame-kirjastossa. Erityisesti MP3 -formaatille ei ole riittävää tukea, jotta sitä kannattaisi käyttää, ja se saattaa johtaa ohjelman kaatumiseen erityisesti Linux -käyttöjärjestelmissä. Oletusarvona äänitiedostojen odotetaan olevan ”sounds”
-kansiossa.

Fonttitiedostoja käytetään käyttöliittymän tekstien piirtämiseen. Fonttitiedostoja tuetaan TrueType (.ttf) -muodossa, jolle löytyy tuki Pygame-kirjastosta. Oletusarvona fonttitiedostojen odotetaan olevan ”fonts” -kansiossa.

Tekstimuotoisia asetustiedostoja käytetään pelin hahmojen ja karttojen rakentamiseen. Asetustiedostot ovat .txt -muodossa, jotta taataan hyvä käyttöjärjestelmätuki ja kaikki tekstieditorit pystyvät helposti muokkaamaan niitä. Asetustiedostoja on yksi hahmoille ja yksi kartoille. Hahmoasetustiedoston oletusarvoinen nimi on ”character_config.txt” ja kartta-asetustiedoston oletusarvoinen nimi on ”map_config.txt”. Näiden odotetaan oletusarvoisesti löytyvän samasta kansiosta ohjelman kooditiedostojen tiedoston kanssa.

Asetustiedostojen lukemisen suorittaa ConfigFileReader -luokan instanssi. Se luo lukemastaan tiedostosta Pythonin listan seuraavalla tavalla.

Rivit, jotka alkavat kahdella kauttaviivalla ”//”, ovat kommentteja ja niitä ei oteta huomioon. Samoin tyhjät rivit ohitetaan.

Asetustiedoston ensimmäisen rivin oletetaan sisältävän tiedostomuodon hyväksymistä varten merkkijono ”DISCO KNIGHTS Config”. Muussa tapauksessa tiedoston sisältöä ei tunnisteta ja lukija näyttää virheilmoituksen.

Asetustiedosto koostuu osioista, ja asetustenlukija rakentaa koko tiedostosta Pythonin listan, jossa jokainen osio on yksi elementti. Jokainen osio puolestaan on sanakirjarakenne. Kokonaisuudessaan muodostetaan siis lista sanakirjoja.

Asetustiedoston osio alkaa ristikkomerkillä ”#”, jonka jälkeen samalle riville kirjoitetaan osion tunniste. Jokaisesta osiosta luodaan listaan elementti, joka on Pythonin sanakirja. Ensimmäinen sanakirjan avain-arvo-pari tehdään tunnisteesta. Jos esimerkiksi osion tunniste olisi ”Map”, tehtäisiin siitä sanakirjaan seuraava avain-arvo-pari:

```
”id”: ”map”
```

Osiossa olevat tiedot kirjoitetaan omille riveilleen ja niiden oletetaan olevan kirjoitettuna tiedostoon seuraavassa perusmuodossa:

```
asetus: arvo
```

Lukija jakaa kaksoispisteen avulla rivin osat, ja ensimmäinen osa valitaan sanakirjaan avaimeksi, ja jälkimmäinen osa arvoksi. On myös mahdollista asettaa yhdelle asetukselle useita arvoja, jolloin ne erotellaan pilkulla toisistaan:

```
asetus: arvo, arvo, arvo, arvo
```

Jos lukija tunnistaa rivillä pilkun, tehdään arvosta lista, ja listaan sijoitetaan elementeiksi kaikki pilkulla toisistaan erotetut arvot. Yllä olevasta tehtäisiin sanakirjaan seuraava elementti avain-arvo-pari:

```
”asetus”: [”arvo”, ”arvo”, ”arvo”, ”arvo”]
```

Yhdelle asetukselle voidaan luoda useita arvoja myös toisella tavalla niin, että myös jokainen arvo voi sisältää useita ala-arvoja. Tämä voidaan toteuttaa kahdella tavalla. Molemmissa tapauksissa asetuksen nimen kanssa samalle riville kirjoitetaan kaksoispisteen jälkeen avaava kaarisulku, jokainen arvo kirjoitetaan ala-arvoineen uudelle riville, ja asetus lopetetaan sulkevalla kaarisululla, joka kirjoitetaan uudelle riville. Ensimmäinen tapa esittää arvoja on luoda jokaiselle arvolle oma avaimensa, ns. ”ala-asetus”. Tällöin avain erotettaisiin tutulla tavalla kaksoispisteellä arvoista, joita voisi jälleen olla useita pilkulla erotettuina. Tämä on hyödyllistä esimerkiksi hahmojen kuvatiedostoja määriteltäessä, kun voidaan erotella jokaisen kävelysuunnan kuvat arvoina omiin ala-asetuksiinsa yhden yläasetuksen alle. Seuraavassa havainnollistava esimerkki. Jos asetustiedostoon kirjoitettaisiin seuraava:

```
asetus: {
    avain: arvo, arvo
    avain: arvo, arvo
}
```

Tekisi lukija siitä seuraavan avain-arvo-parin, jossa arvo on sanakirja itsessään:

```
”asetus”: {
    ”avain”: [”arvo”, ”arvo”],
    ”avain”: [”arvo”, ”arvo”]
}
```

Toinen vaihtoehto olisi listata vain ala-arvot ilman avaimia, jolloin ne eroteltaisiin välilyönnillä. Tämä on erityisen käytännöllistä erityisesti karttoja muokattaessa, kun niiden ruutuja määritellään. Esimerkiksi, jos kirjoitettaisiin näin:

```
asetus: {
    a b
    c d
}
```

Tehtäisiin siitä seuraava avain-arvo-pari, jossa arvo on listarakenne:

```
”asetus”: [
    [”a”, ”b”],
    [”c”, ”d”]
]
```

Rivien lukemista ja uusien asetuksien lisäämistä samassa osiossa jatketaan, kunnes tunnistetaan seuraava lohko eli rivi, joka alkaa ristikkomerkillä. Tällöin osion sanakirja suljetaan, ja luodaan uusi sanakirja seuraavasta osiosta.

Asetustenlukija osaa käsitellä ylimääräiset välilyönnit ja tyhjän tilan, jolloin voidaan helpottaa luettavuutta esimerkiksi sisennysten ja tyhjien rivien avulla.

Asetustenlukija on suunniteltu ensin rakentamaan tiedostosta tietorakenteen katsomatta tarkemmin sisältöön, jotta uusien ominaisuuksien ja asetusten lisääminen olisi helppoa. Tästä syystä on myös huomattava, että se ei ota kantaa siihen, onko asetuksen arvo luku vai tekstiä, vaan kaikkea käsitellään merkkijonoina, ja muutokset tulee tehdä myöhemmin. Noudattaen yllä mainittuja muotoilusääntöjä / syntaksia voidaan lisätä uusia asetuksia, jotka lisätään tietorakenteeseen, ja niitä voidaan sen jälkeen käyttää esimerkiksi kartan ja hahmot rakentavissa metodeissa. Myöskään ei tarkisteta erityisesti, että kaikkia asetuksia käytettäisiin. Tiedoston lukeminen on näin eritelty täysin sen sisällön käyttämisestä, ainoana poikkeuksena ylimmän rivin tunnisteen tarkistaminen. Näin ollen samaa koodia voitaisiin helposti käyttää vaikka seuraavissa projekteissa. Tämä helpotti myös suuresti työskentelyä projektin aikana, kun asetustenlukija sopeutui automaattisesti uusiin lisättyihin ominaisuuksiin.

Kartat ja hahmot rakennetaan asetustiedostojen perusteella. Lukija luettua tiedoston se antaa tuottamansa tietorakenteen eteenpäin rakennusmetodeille, jotka käyttävät asetuksia olettaen, että ne on määritelty tiedostoon seuraavalla tavalla.

Hahmot

Hahmoille määritellään nimi, joukkue, maksimielämä, liikkumisen kantama yhdellä vuorolla, alkusuunta, onko se tekoälyn kontrolloima ja lisäksi sen kuvatiedostot. Kävelyanimaatioon käytettäviä ”walk_sprites” kuvatiedostoja ei ole välttämätöntä määritellä, jolloin normaali seisova hahmo liukuu kartalla kävellessä. Esimerkiksi ensimmäinen oletushahmo on määritelty osiona seuraavalla tavalla (kaikkia kävelyanimaation kuvia ei ole esitetty tässä).

```
#Character
name: A
team: 1
max_health: 100
move_range: 3
facing: right
is_ai: false
stand_sprites: {
          up: hero1_up.gif
          down: hero1_down.gif
          left: hero1_left.gif
          right: hero1_right.gif
}
walk_sprites: {
```

```
            up: hero1_walk1_up.gif, hero1_walk2_up.gif
            down: hero1_walk1_down.gif, hero1_walk2_down.gif
            left: hero1_walk1_left.gif, hero1_walk2_left.gif
            right: hero1_walk1_right.gif, hero1_walk2_right.gif
}
```

Hahmoille määritellään käytettävissä olevat toiminnot seuraavalla tavalla jokainen toiminto omassa osiossaan. Jokaiselle hahmolle luetaan maksimissaan kolme toimintoa.

```
#Action
character: A
type: damage
strength: 35
max_range: 1
name: Hit
sound: sword.ogg
```

Jokaiselle toiminnolle määritellään hahmo, jolle se kuuluu, toiminnon tyyppi, vahvuus, nimi ja kantama. Vapaavalintaisesti voi myös määritellä ääniefektin, joka toistetaan, kun toimintoa käytetään. Käytettävissä on seuraavat toimintotyypit:

damage:
Vähentää kohteena olevan hahmon elämää vahvuutena määritellyllä luvulla.

heal:
Lisää kohteena olevan hahmon elämää vahvuutena määritellyllä luvulla.

stun:
Tainnuttaa kohteena olevan hahmon vahvuutena määritellyksi luvuksi vuoroja.

buff:
Muuttaa kohteena olevan hahmon toimintojen vahvuutta määritellyllä kertoimella seuraavaksi vuoroksi, voi siis olla joko nollasta yhteen, jolloin se vähentää toimintojen vahvuutta, tai yhdestä ylöspäin, jolloin se lisää vahvuutta. Toiminnon nimen tulisi kuvastaa suuntaa. Tässä ei ole rajoitettu negatiivisten lukujen käyttöä, jolloin halutessa voitaisiin esimerkiksi tehdä toimintoja, jotka ”hämäävät” kohdehahmoa tekemään päinvastaisia asioita, kun oli tarkoitus.

Hahmoasetustiedostoon on valmiiksi määritelty kolme pelaajan hahmoa ja kolme tekoälyn kontrolloimaa hahmoa.

Kartat

Karttoja varten määritellään kartta-asetustiedostossa ruututyypit, esineet/objektit ja itse kartat.

Ruututyypeille määritellään nimi, lyhenne, voiko niissä kävellä (true tai false) ja kuvatiedosto seuraavasti:

```
#Squaretype
name: water
short: w
```

```
walkable: false
sprite: water.gif
```

Esinetyyppejä taas voidaan määritellä seuraavasti:

```
#Object
name: rock
short: r
sprite: rock.gif
offset_x: 0
offset_y: -12
```

Esineille määritellään nimi, lyhenne, kuvatiedosto ja kuvan piirtämisen siirros. Siirrosarvoilla saadaan asetettua esineet haluttuun kohtaan ruutuun, sillä esineiden koko vaihtelee.

Ruutu- ja esinetyyppien määrittelyjen jälkeen voidaan määritellä itse kartta. Kartalle annetaan järjestysluku, leveys, korkeus, hahmojen lähtöpisteet sekä itse ruudut. Nämä määritellään seuraavasti:

```
#Map
index: 4
width: 13
height: 7
team1_start: {
    2 2
    2 3
}
team2_start: {
    10 3
    10 4
}
squares: {
    g.t w w w w  w  g  bh g  w w w g.t
    w   w w w w  w  bv w  bv w w w w
    w   w g g g  g  g  w  g  g g w w
    w   w g g g  g  g  g  g  g g w w
    w   w g g g  w  g  g  g  g g w w
    w   w w w bv w  bv w  w  w w w w
    g.t w w w g  bh g  w  w  w w w w
}
```

Yllä olevassa lähtöpisteiksi asetetaan esimerkiksi joukkueelle 1 koordinaatit (2,2) ja (2,3), jolloin kartalle lisätään vain kaksi hahmoa joukkueesta 1, vaikka niitä olisi hahmotiedostossa enemmän. Lähtöpisteet määritellään siis karttakohtaisesti. Ruutujen määrittelyssä käytetään aiemmin määriteltyjä lyhenteitä. Ensin kirjoitetaan haluttu ruututyyppi, ja sen jälkeen haluttaessa pisteellä erotettuna esine. Esimerkiksi yllä vasemman ylänurkan ruutuun on määritelty ruohon päälle puu.

Kartta-asetustiedostoon on valmiiksi määritelty neljä ruututyyppiä:

Water (w)
Vesiruututyyppi, jossa ei voi kävellä

Grass (g)
Ruohoruututyyppi, jossa voi kävellä

Bridge_hor (bh)
Vaakasuuntainen siltaruututyyppi, jossa voi kävellä

Bridge_ver (bv)
Pystysuuntainen siltaruututyyppi, jossa voi kävellä

Ja kaksi esinetyyppiä:

Rock (r)
Kivi

Tree (t)
Puu, jonka kuva on hieman läpinäkyvä

Näistä on kasattu valmiiksi neljä karttaa.

8. Testaus

Ohjelmaa testattiin sekä yksikkötesteillä että manuaalisesti kokeilemalla mahdollisimman monia erilaisia tilanteita. Erityisesti käyttöliittymän toimintaa vaativia asioita testattiin manuaalisesti, ja niitä, jotka eivät vaadi käyttöliittymän käynnistämistä, testattiin yksikkötesteillä. Yksikkötestit löytyvät test.py -tiedostosta ja niiden tarkka toiminta on kommentoitu koodiin.

Yksikkötesteillä testattiin seuraavia:

Etäisyyden sisällä olevien ruutujen etsintä

Tätä algoritmia testattiin yksikkötesteillä luomalla testikartta, jolle asetettiin erityyppisiä ruutuja sekä testiesine ja hahmoja. Kutsumalla metodeita eri tilanteita simuloivilla ehdoilla testattiin, että oikea määrä ruutuja palautetaan, kun haetaan koordinaatit jonkin etäisyyden sisällä eri tilanteissa. Yksikkötestien lisäksi tätä testattiin myös manuaalisesti Map.get_range_count_map() -metodin avulla. Tämä metodi palauttaa merkkijonoesityksenä kartan muodossa ruutuolioille asetetut etäisyydet lähtöpisteestä, ja tulostamalla tämä voidaan manuaalisesti verrata algoritmin toimintaa haluttuun. Sekä yksikkötestit että manuaalinen testaus menivät läpi, ja algoritmi toimii oletetulla tavalla.

Lyhimmän polun etsintä

Lyhimmän polun etsivää algoritmia testattiin myös sekä yksikkötesteillä että manuaalisesti. Yksikkötestaus kuuluu samaan testitapaukseen edellisen algoritmin kanssa. Luotiin siis testikartta, jolla haettiin lyhintä polkua, ja verrattiin saadun polun pituutta oletetun polun pituuteen. Manuaalisesti tätä metodia voitiin testata tulostamalla kartta Map.get_simple_map()

-apumetodin avulla ja etsimällä sitten käsin lyhin reitti pisteiden välillä, ja vertaamalla tätä reittiä Map.get_shortest_path() -metodin palauttamaan polkuun. Sekä yksikkötestaus että manuaalinen testaus osoittivat, että algoritmi toimii halutulla tavalla eri tilanteissa, ja osaa käsitellä myös tilanteet, joissa polkua kohteeseen ei ole ollenkaan.

Hahmojen liikkuminen

Hahmojen liikkumisen testaaminen yksikkötesteillä testasi apumetodeita, jotka kääntävät ja liikuttavat hahmoja kartalla ruutu kerrallaan, ja sitten näihin yhdistettynä lyhimmän polun etsinnän kanssa liikkumista. Tätä testattiin myös manuaalisesti yllä mainittujen kartantulostusmetodien avulla. Testit osoittivat, että metodit toimivat oletetulla tavalla myös tilanteissa, joissa hahmon edessä on esteitä, jolloin liikkumisen ei pitäisi olla mahdollista.

Toimintojen vaikutukset

Toimintojen vaikutuksia testattiin yksikkötesteillä luomalla jälleen testikartalle testihahmo, ja kutsumalla metodeita, joiden tulisi vahingoittaa hahmoa tai aiheuttaa muu vaikutus, ja sitten vertaamalla olion attribuutteja oletettuihin.

Vuorojenhallinta

TurnController -luokan instanssi hallitsee pelin vuoroja, ja sitä testattiin yksikkötesteillä niiltä osin, kun ei vaadita graafisen käyttöliittymän toimintaa. Yksikkötestit varmistivat, että hahmojen lisääminen ja kontrollerin nollaaminen toimivat oikein.

Tekoäly

Yksikkötesteillä testattiin tekoälyn osalta erityistilanteita, kuten sellaisia, joissa kaikki toisen joukkueen hahmot ovat saavuttamattomissa, jolloin mitään kohdetta ei pitäisi valita, ja metodien pitäisi palauttaa metodista riippuen esim. False. Testattiin myös, että saadaan oikea lyhin polku hahmolle valittuun kohteeseen.

Asetustenlukija

ConfigFileReader -luokkaa testattiin yksikkötesteillä mahdollisten virheellisten syötteiden osalta. Näihin kuuluivat esimerkiksi sulkemattomat sulut, väärä ensimmäisen rivin tunniste, tunnistamattomassa muodossa oleva sisältö. Lukijan tulisi heittää ConfigFileError -virhe, kun jokin lukemisessa menee pieleen, ja yksikkötesteillä testattiin, että tämä virhe saatiin aikaiseksi, ja että oikeanlainen syöte ei sitä aiheuta.

Yksikkötestausta tehtiin jossain määrin lähes koko kehityksen ajan, ja monet testeistä olivat hyödyllisiä kehityksen keskellä, mutta eivät enää soveltuneet graafisen käyttöliittymän kanssa toimimiseen. Testejä lisättiin ja poistettiin kehityksen aikana, ja yritin mahdollisimman usein ajaa niitä läpi, jotta en aiheuttaisi muuttaessani asioita sivuvaikutuksia muualle ohjelmaan. Tämä osoittautuikin hyväksi käytännöksi, koska se toi esiin monia kohtia, joissa ohjelman ajaminen pelkästään ei olisi käyttänyt kaikkia metodeja, mutta yksikkötestien avulla huomasin näitä sivuvaikutuksia, ja sain korjattua asioita ennen kuin ne kasaantuivat.

Muu testaaminen

Yksikkötestauksen lisäksi ehkä suuremmassa roolissa oli kuitenkin käsin testaaminen, erityisesti graafisen käyttöliittymän kehityksessä. Kun GUI oli pystyssä, pystyi sen avulla helposti näkemään, mitä tapahtuu näytöllä, ja verrata sitä siihen, mitä pitäisi tapahtua. Kaikkia yllä lueteltuja osioita tuli jo pakostikin testattua jatkuvasti kehityksen aikana, kun näytöllä näkyi pelin tapahtumat. Etäisyyksien etsinnän tulos näytetään pelissä kartalla, joten pystyi helposti todentamaan, että oikeat koordinaatit löydettiin. Lyhimmän reitin etsintää hyödynnetään hahmojen kävelyssä, joten oikean polun löytyminen selvisi hahmoa seuraamalla. Samalla myös hahmojen liikkuminen testautui. Tekoälyn toimintaa pystyi tarkemmin myös testaamaan seuraamalla tekoälyhahmojen liikkumista ja toimintaa ruudulla. Suunnitelman mukaisesti käytin myös runsaasti väliaikaisia tulostuskäskyjä metodien koodissa, jolloin niiden sisäistä toimintaa pystyi seuraamaan ja selvittämään, missä kohdassa ongelmat tapahtuivat. Tekoälyn osalta lisäsin käskyjä, jotka tulostivat hahmojen sijoitukset etäisyyksien ja jäljellä olevan elämän perusteella, josta pystyin manuaalisesti vertaamaan laskettiinko nämä oikein. Hahmojen tila ja vuorot näkyvät pelin aikana ruudun ylälaidassa, joten siitä oli helppo seurata tapahtuuko oikeita asioita pelin edetessä toimintojen vaikutusten vuorojenhallinnan osalta, esimerkiksi vuoronlopettamisnappia painettaessa. Käyttöliittymän kautta tapahtuva testaus oli luonnollisesti jatkuvaa, koska heti käyttöliittymän kasauksen jälkeen alkoi nähdä näytöllä taustalogiikan toimintaa, jolloin ns. testaamista oli vaikea välttää.

Suurta osaa koodista oli vaikea testata yksikkötesteillä, joten sen testaus riippui kokonaan graafisen käyttöliittymän kautta tapahtuvasta kokeilusta. Tässä yritettiin olla mahdollisimman järjestelmällisiä siinä mielessä, että pyrittiin aiheuttamaan mahdollisimman monia erilaisia tilanteita, joihin pelin aikana voitaisiin päätyä, koska jotkin virheet saattavat nousta vasta hyvin erityisessä ja ehkä harvinaisessa tilanteessa. Tällä tavalla havaittiinkin alussa esimerkiksi tilanteita, joissa hahmot jäivät toisten hahmojen taakse piiloon, ja lyhimmän polun etsintä ei onnistunut, vaan jäi rullaamaan loputonta silmukkaa.

Suunnitelman mukaisesti ennen käyttöliittymän testausta käytettiin paljon apumetodeita, jotka tulostivat tietoja näkyviin, ja paljon yksikkötestejä. Jossain määrin ei-suunnitellusti yksikkötestit osoittautuivat kuitenkin odotettua kätevämmiksi, kun useita samoja testitapauksia ajoi usein läpi, ja näki mitä sivuvaikutuksia omat muutokset ja virheet aiheuttivat.

Kokonaisuudessaan testaus meni lähestulkoon suunnitelman mukaisesti lukuun ottamatta sitä, että alussa ennen graafista käyttöliittymää samoja yksikkötestejä ajettiin kehityksen aikana, kun suunnitelmaa tehdessä ajattelin sen olevan enemmän vain vaiheen lopussa tapahtuvaa.

Ilman graafista käyttöliittymää toimivat yksikkötestit on kommentoitu askeleittain testitiedostossa. Suuri osa koodista vaatii kuitenkin graafisen käyttöliittymän osien alustamista, joten niitä on testattu manuaalisesti.

Tekoälyä testattiin reilusti myös niin, että asetettiin kaikki hahmot tekoälyn hallintaan, ja annettiin sen pelata itseään vastaan lukuisia kertoja.

Ohjelmaa on testattu Mac OS X Yosemite, elementary OS (linux) ja Windows 8.1 käyttöjärjestelmillä, Python versiolla 3.4 (64bit) ja Pygamen 64bit versiolla (linkki viitteissä).

9. Ohjelman tunnetut puutteet ja viat

Seuraavassa on lueteltu tuntemani puutteet ja viat sekä asioita, joita muuttaisin, jos jatkaisin projektia pidemmälle.

Peliin ei ole lisätty animaatiota hyökkäyksille tai kuolemalle
Hahmoille on piirretty vain kävelyanimaatiot kaikkiin suntiin, ja niillä ei ole animaatioita hyökkäyksille tai kuolemalle. Toiminnoille on kuitenkin annettu käyttäjän määriteltävissä oleva ääniefekti. Tämä johtuu lähinnä siitä, että grafiikoiden piirtäminen vie huomattavasti aikaa, joka ei varsinaisesti liity ohjelmointiin. Tämä olisi kuitenkin helppo lisätä peliin samaan tapaan kuin kävelyanimaatio sillä erotuksella, että nimaation käytäisiin läpi vain kerran. Luettaisiin kuvat ensin asetustiedostosta samaan tapaan kuin kävelyanimaation kuvat, kirjoitettaisiin sitten CharacterView -luokkaan uusi metodi get_next_action_frame(), joka valitsisi seuraavan kuvan animaation vaiheista niin kauan kuin kuvia olisi jäljellä, sitten lisättäisiin ChacacterView-luokan update() -metodiin esimerkiksi ehto ”if self.character.action_animation”, jonka toteutuessa kutsuttaisiin edellä tehtyä get_next_action_frame() -metodia ja vaihdettaisiin CharacterView-olion valittu kuva niin kauan kun animaation vaihteita on jäljellä. Kun ei enää saada seuraavaa vaihetta, vaihdetaan kuva takaisin normaaliin seisontakuvaan oikeaan suuntaan. Animaatio voitaisiin pelaajan osalta laukaista jo muutenkin toiminnot tunnistavissa kohdissa GameEventHandlerin handle_click() ja handle_event() -metodien alle asettamalla esimerkiksi vain edellä mainittu ”character.action_animation” -attribuutti todeksi, jolloin update()-metodi tunnistaisi animaation automaattisesti. Animaation viimeisen kuvan jälkeen vain asetettaisiin tämä taas epätodeksi. Animaatiot olisi siis helppo lisätä myös koodiin, jos niitä piirtäisi lisää.

Ei nappia, josta voisi peruuttaa toiminnon suorittamisen ja liikkua vielä toiminnon valinnan jälkeen, jos ei vielä liikkunut
Kun pelissä valitsee jonkin toiminnon, ei siitä voi enää palata takaisin liikkumaan, vaikka ei olisi vielä liikkunut. Muuttaisin tämän niin, että lisäisin napin, jolla voisi peruuttaa toiminnon valinnan ja näytettäisiin jälleen ruudut, joihin pystyy liikkumaan. Nyt toiminnon valinnan jälkeen on mahdollista ainoastaan joko käyttää toiminto tai lopettaa vuoro.

Karttaobjektien taakse kävellessä ne voisivat muuttua läpinäkyviksi, nyt sitä ei tapahdu
Monesti peleissä, joissa kuvakulma on kiinteä, esineet muuttuvat läpinäkyviksi, kun hahmo kävelee niiden taakse. Nyt tämä ominaisuus puuttuu, ja jos esineen kuva on täysin läpinäkymätön, on sen taakse mahdoton nähdä.

Testauksen aikana yksi tuntematon kaatuminen
Tekoälyn testauksen aikana tapahtui yksi selvittämätön kaatuminen, jota ei saatu toistettua kertaakaan sen jälkeen. Kaatuminen tapahtui, kun tekoälyn annettiin pelata itseään vastaan niin, että kaikki hahmot oli asetettu tekoälyn hallintaan. Yhden hahmon kuoltua tapahtui jotain, ja ohjelma jäätyi tuntemattomasta syystä. Tilannetta toistettiin sen jälkeen ainakin kymmeniä kertoja, mutta kaatumista ei saatu toistettua uudelleen, joten sitä ei pystytty selvittämään.

Alla on vielä lueteltuna asioita, jotka eivät välttämättä ole niinkään puutteita tai vikoja, vaan lähinnä asioita, joita muuttaisin ja edistäisin, jos jatkaisin projektia pidemmälle.

Karttaobjekteille mahdollisuus kävellä niistä läpi
Ohjelma ei tällä hetkellä mahdollista sitä, että objektien läpi voisi kävellä, ja että tämän voisi määritellä asetustiedostossa. Tämän voisi muuttaa lisäämällä objektityypeille samalla tavalla tämän ominaisuuden kuin ruututyypeille, ja muuttamalla sen myös metodeihin, jotka etsivät etäisyyden sisällä olevia ruutuja ja lyhintä polkua.

Lisäisin ruututyypeille ominaisuuden, että hahmo ei voi suorittaa toimintoa niissä
Lisäisin ruututyyppien ominaisuuksiin sellaisen ominaisuuden, että hahmo ei voi suorittaa toimintoa niissä, esim. koska maasto on liian vaikea kulkea tms. Tämä voitaisiin vain lisätä SquareType -luokkaan ja kartta-asetustiedostoon, ja sitten vain tarkastaa ennen toiminnon suorittamista esimerkiksi GameEventHandlerissa, että hahmo ei ole sellaisessa ruudussa, jossa ei voi suorittaa toimintoa.

Pelin aikana ei näytetä, jos johonkin hahmoon on käytetty ”buff” toimintoa ja oletettavissa on, että sen toiminnot ovat esim. vahvempia seuraavalla vuorolla
Tämän tiedon voisi asettaa näkyviin hahmojen tietojen kanssa samaan tapaan kuin nähdään, että jokin hahmo on tainnutettu. Toinen vaihtoehto olisi piirtää hahmon pään päälle jokin indikaattori, jota voitaisiin myös käyttää muissa tapauksissa, kuten tainnuksissa olon ilmaisemiseen ja myös vuorossa olevan hahmon ilmaisemiseen.

Kaikki Statet voisi tallettaa johonkin muuhun tietorakenteeseen kuin suoraan attribuuteiksi, esim. sanakirjaan
Nyt ohjelman mahdolliset tilat on asetettu StateManagerin attribuuteiksi, mikä on melko kiinteä ratkaisu, ja tämä voitaisiin muuttaa niin, että tilat talletettaisiin esimerkiksi sanakirjaan avain-arvo-parina ”nimi”: State-olio. Näin tiloja olisi helpompi ja dynaamisempi käsitellä ilman kiinteitä määrityksiä.

Kartan liikuttaminen nuolinäppäimillä
Karttaa ei voi liikuttaa näytöllä, eli jos tekisi tarpeeksi ison kartan, se menisi reunusten yli eikä laitoja välttämättä näkisi. Kartan pitäisi olla yli 16x16 ruutua, jotta se menisi osittain piiloon, eli melko suuri. Tästä syystä asetustenlukijaan on asetettu virheilmoitus, jos pelaaja yrittää tehdä kartan, jonka leveys tai korkeus on suurempi kuin 16. Jos karttaa voisi liikuttaa näytöllä nuolinäppäimillä, ei tällaista kokorajoitetta tarvitsisi tehdä, koska jokainen kartan osio olisi saatavilla karttaa siirtämällä. Tämän toteuttamiseksi tulisi lisätä esimerkiksi MapView -luokalle attribuuteiksi siirrokset x- ja y-suunnissa, ja tämä siirros lisättäisiin piirrettäessä piirtokoordinaatteihin. CharacterView ja MapObjectView lukisivat saman siirroksen MapView-oliolta, koska hahmot ja objektit on liitetty karttaan muutenkin, ja ottaisivat tämän myös huomioon piirtäessään. Sitten asetettaisiin vain GameEventHandleriin handle_key()-metodin alle ehdot kaikille nuolinäppäimille (esim. ylöspäin nuoli on pygame.K_UP) ja joko lisättäisiin tai vähennettäisiin x- tai y-suunnan siirrosta sen mukaan mitä nuolinäppäintä painettiin.

Manuaalinen kartan indeksi asetustiedostossa ja manuaaliset mitat
Karttatiedostossa tulee asettaa kartalle indeksi/järjestysluku ja mitat manuaalisesti. Muuttaisin nämä tunnistettavaksi automaattisesti asetustenlukijassa.

Virheilmoitukset kartan rakennuksessa tarkemmaksi
Asetustiedostot lukeva ConfigFileReader osaa antaa virheilmoituksia ja kertoa suunnilleen, missä virhe tapahtui. Monien virheiden kohdalla se osaa kertoa täsmälleen mistä virhe johtui ja millä rivillä se tapahtui, mutta erityisesti kartan rakentamisessa

lisäisin siihen tarkempia ilmoituksia siitä, miksi virhe tapahtui. Nyt se osaa kertoa missä kartassa virhe tapahtui, mutta ei välttämättä tarkasti miksi.

Hahmojen kuvatiedostot voisivat olla spritesheetissä eikä yksittäisinä kuvina
Hahmoille on nyt asetettu kaikki animaatioiden vaiheet/kuvat yksitellen. Jos jatkaisin projektia, muuttaisin tämän niin, että ohjelmalle voisi syöttää yhtenä kuvana kaikki yhden hahmon spritet tai ainakin kaikki yhden suunnan kuvat, ja ohjelma osaisi jakaa kuvan sitten yksittäisiin kuviin tiedettyjen mittojen mukaan esimerkiksi Pygamen kuvankäsittelymetodien avulla.

10. 3 parasta ja 3 heikointa kohtaa

Parhaat kohdat

StateManager - State - EventHandler -rakenne sekä näkymien erottelu
Olen hyvin tyytyväinen tähän ohjelman perusrakenteeseen, koska se selkeästi erottelee osuudet toisistaan ja jo kehityksen aikana huomasi, että se on hyvin mukautuva. Tiloja (esim. valikoita) oli helppo lisätä sen avulla ja niiden välillä liikkuminen on helppoa. Tätä rakennetta voin varmasti käyttää uudelleen myöhemmissä projekteissa ilman kovin suuria muutoksia, koska sen yliluokat on määritelty melko yleisellä tasolla, ja abstrakteja luokkia ja perintää käytetään sitten hyödyksi. Kun olin tehnyt tämän rakenteen valmiiksi, oli kehitys sen jälkeen hyvinkin sujuvaa ja oli helppo aina tietää minne mikäkin koodinpätkä piti laittaa loogisesti. Mielestäni myös karttojen, objektien ja hahmojen näkymien erottelu niistä itsestään on toimiva ratkaisu, ja periaatteessa kullakin näkymäluokalla on olennaisimpina metodit update ja draw, jotka päivittävät ja piirtävät kyseisen näkymän oman taustalogiikkaolionsa perusteella. Tämä rakenne oli myös selkeä kehitettäessä, joten oli helppo pitää kirjaa siitä, missä mikäkin asia oli ja mihin mitäkin piti seuraavaksi koodata. Pääsilmukassa voitiin sitten vain pyytää StateManageria kutsumaan nykyisen Staten update- ja draw-metodeita.

Ohjelman muokattavuus / konfiguroitavuus
Peli on mielestäni hyvin pitkälle muokattavissa ulkoisten tiedostojen kautta. Kaikki kartat, hahmot ja toiminnot määritellään asetustiedostoissa. Ulkoasua olisi helppo muuttaa halutessaan aivan eri tyyliseksi. Itse käyttöliittymän nappien yms. kuvatiedostojen polut on määritelty myös ”constants” -moduulissa, jolloin niitäkin voitaisiin muokata, samoin musiikit ja ääniefektit ovat muokattavissa. Peliin on myös helppo lisätä uusia käyttäjän muokattavia asetuksia, koska asetustenlukija on toteutettu niin, että ainoastaan tiedoston rakenteella on väliä, eikä varsinaisesti asetusten nimillä tai arvoilla. Lukija vain tekee tiedostosta tietorakenteen, joka sisältää kaikki sen asetukset, ja sitten myöhemmin tutkitaan karttoja ja hahmoja rakennettaessa, että oikeat asetukset löytyivät.

Suorituskyky
Tämä on omasta näkökulmastani erityisen hyvä onnistuminen, kun vertaan projektin aikana tapahtunutta oppimista ja ensimmäisen toteutuksen versiota uudelleenrakentamisen jälkeiseen lopputulokseen. Ensimmäisessä toteutuksessa tein liikaa asioita pääsilmukassa ja toistin paljon kuvien latauksia ja muita asioita, joita sain vähennettyä erittäin merkittävästi uudella suunnitelmalla. Esimerkkinä voisi sanoa, että ensimmäisellä kerralla tehtiin jokaisella ruudunpäivityskerralla satoja ja

pahimmillaan jopa pari tuhatta ”blit” -operaatiota kaikkien elementtien piirtämisessä ruudulle, ja suorituskyky oli todella heikko, vaikka ruudulla päivitettiin vieläpä ainoastaan osuudet, jotka olivat muuttuneet. Uuden rakenteen eli tilojen ja näkymien avulla sekä muiden parannusten kautta sain vähennettyä jokaisella ruudunpäivityskerralla suoritettavia ”blit” -operaatioita niin, että niitä tapahtuu pelin aikana keskimäärin vain noin 15-25 eli hyvinkin vähän. Nyt pelin ruudunpäivitysnopeutta pitää jopa rajoittaa, jotta esim. hahmot eivät liiku liian nopeasti, ja päästäisiin aina Pygamelle tyypilliseen 60fps nopeuteen, jos sitä ei rajoitettaisi.

Heikot kohdat

Luokkien rajapinnat
Luokista toisiin viitatessa on käytetty melko paljon pitkiä viittauksia, kuten esim. ”self.state_mgr.main_menu.menu.options”, eli mennään useiden rakenteiden kautta näin. Toisaalta niissä näkyy suoraan, minne ohjelman rakenteessa viitataan, mutta ketjuista tulee hyvinkin pitkiä. Pythonissa on kuitenkin hyvä tapa (”pythonic way”) viitata suoraan attribuutteihin niitä hakiessa ja asetettaessa, attribuutit eivät ole yksityisiä. Ohjelman laajuuden huomioon ottaen rajapinnat ovat tärkeässä roolissa, joten niiden tulisi olla selkeät ja helppokäyttöiset. Nykyinen ratkaisu siis kuitenkin toimii juuri halutulla tavalla lopputulokset kannalta, mutta mietin kyllä monesti, olisiko tähän joku selkeämpi ratkaisu.

Lyhimmän polun etsintä olettaa, että ruutujen etäisyysluvut ovat valmiiksi oikein
Lyhimmän polun etsivä metodi olettaa, että sitä ennen on kutsuttu Map.set_in_range() metodia, joka asettaa ruutujen etäisyysluvut lähtöpisteestä. Tässä pelissä oletuksesta ei ole haittaa ja se lähinnä vähentää ylimääräistä algoritmien suoritusta, mutta jos oletusta ei ota huomioon, saattaa lyhimmän polun etsivä algoritmi tuottaa mahdollisesti väärän lopputuloksen. Muuttaisin tämän niin, että oletusta ei tehdä, ja saattaisin myös erottaa etäisyyslukujen asetuksen set_in_range() metodista, vaikka se aiheuttaisikin vähän suuremman aikakompleksisuuden kokonaisuudessaan, koska se helpottaisi koodin uudelleen käyttämistä muualla ja erottelisi osatehtävät selkeämmin toisistaan. Nykyinen ratkaisu toimii siis nopeammin, mutta näin pienellä ruudukolla suoritusajan kasvu ei varmastikaan aiheuttaisi huomattavaa muutosta. Isommalla ruudukolla tosin voitaisiin valita myös Lee algoritmin sijaan kokonaan toinen algoritmi lyhimmän polun etsintää varten.

Tekoäly voisi olla monimutkaisempi
Tekoäly toimii sujuvasti pelin aikana tilanteesta riippumatta ja osaa mukautua esimerkiksi hahmoille annettuihin erilaisiin toimintoihin, mutta olisin halunnut loppupelissä tehdä siitä hieman monimutkaisemman. Se olisi voinut esimerkiksi verrata omaa hahmoaan vastustajan hahmoihin, ja olla hyökkäämättä itselleen liian vahvojen vastustajien kimppuun ja pyrkiä sen sijaan itseensä verrattuna heikkoja kohteita kohti. Pelissä on yksi tilanne, jossa tekoäly ei osaa tehdä rationaalista ratkaisua: kun yhteenkään kohteeseen ei ole polkua, niin se ei osaa valita kohdetta. Jos esimerkiksi kartalla on kapeita paikkoja, ja muut hahmot tukkivat kaikki reitit, odottaa se paikallaan reitin vapautumista, eikä osaa lähestyä tukittua kohtaa. Se lähtee jälleen liikkumaan vuorollaan, kun polku vapautuu, mutta tukkeet pysäyttävät sen hetkeksi.

11. Poikkeamat suunnitelmasta

Suunnitelmasta poikettin olennaisesti luokkarakenteen osalta niin, että suunniteltu luokkarakenne on vain pieni osa toteutunutta rakennetta. Koko ohjelman rakenteen suunnittelua oppi tehdessään projektia, eikä sitä oltu käyty luennoilla läpi, joten suunnitelma ei varsinaisesti ollut riittävä tai kokonainen. Samoin Pygame-kirjaston käytön opettelu tapahtui samaan aikaan kehittämisen kanssa, joten ensimmäinen versio toteutuksesta oli melko huonosti kasattu. Oppimista kuitenkin tapahtui jatkuvasti kehittämisen aikana, joten lopulta rakenne selkeytyi, tai ylipäätään muodostui koko ohjelmaan laajuinen mietitty rakenne.

Ajankäytöllisesti suunnitelma toteutui melko huonosti, koska ensimmäisestä toteutuksesta puuttui kunnollinen rakenne, ja kaikki oli sisäänrakennettuna pääsilmukkaan. Näin ollen suunnittelin rakenteen uudelleen, ja loppujenlopuksi aikaa projektiin tuli käytettyä arviolta yli tuplasti alkuperäiseen suunnitelmaan nähden. Voisi sanoa niin, että alkuperäinen suunnitelma toteutui melko hyvin, mutta sen lopputulos ei ollut halutunlainen, joten tehtiin uusi suunnitelma ja toteutettiin sitten edellisestä opittuna uusi suunnitelma. Näin aikaa käytettiin yhteensä huomattavasti suunniteltua enemmän, mutta lopputuloksesta tuli huomattavasti parempi.

Suunniteltu toteutusjärjestys osui osittain oikeaan. Itse pelin taustalogiikka toteutettiin ensin ja sen jälkeen siirryttiin nopeasti käyttöliittymän opetteluun, kuten suunniteltiin. Ensimmäinen lähes täysin toimiva versio pelistä tuli myös valmiiksi noin checkpointtiin (21.4.) mennessä, mikä oli samassa tahdissa suunnitelman kanssa. Tässä vaiheessa kuitenkin tein uuden suunnitelman ohjelman rakenteen osalta, ja aloitin toisen iteraation koko ohjelman toteutuksesta lähes tyhjästä, minkä jälkeen alkuperäinen aikataulu ei enää pitänyt paikkaansa.

Seuraavassa on kerrottu pienempiä poikkeamia suunnitelmasta.

Suunnitelma ei ottanut huomioon sitä, että ohjelmassa tarvitaan lyhimmän polun etsintää. Siinä esitettiin vain yleisesti, että hahmo liikkuu jotain kohdetta kohti.

Suunnitellut getMovableSquares() ja getActionableSquares() -metodit toteutettiin lopuksi vain yhtenä metodina, koska ne toimivat samalla algoritmilla erona vain jotkin ehdot.

Ohjelmaan on toteutettu suunniteltua isompi valikkosysteemi omien luokkiensa avulla.

On luotu mahdollisuus lisätä kartalle objekteja ruutujen päälle, esimerkiksi puita. Suunnitelmassa oli vain erilaisia ruututyyppejä.

Toimintojen tyypit muuttuivat hieman. Suunniteltu väistäminen poistui, mutta tilalle tulivat parantaminen ja ”buff”, jolla voidaan vahvistaa tai heikentää toisten hahmojen toimintojen vahvuutta hetkeksi. Suunnitelmassa ei ollut omiin joukkuetovereihin kohdistuvia toimintoja.

Toimintoja ei pidetä monikon vaan listan sisällä.

Tekoälyn toiminta muuttui suunnitellusta jonkin verran. Kohde valitaan suunnitelman mukaan hahmojen sijainnin ja jäljellä olevan elämän perusteella, mutta käytettävä toiminto valitaan toisin. Suunnitelmassa valinta perustui enemmän arpomiseen, nyt se on enemmän laskennallinen ja rationaalinen, mutta silti vielä osittain satunnainen.

Tekoäly ei ole yhdistettynä hahmoon, vaan vuorojenhallintaan, ja sitä käytetään suoraan vuorossa olevan hahmon kanssa.

Vuorojenhallintaa varten on myös tehty oma luokkansa, joka eroaa suunnitellusta.

Suunnitelmassa oli 3 erilaista tekoälyhahmoa, mutta toteutettiin vain 2. Tämä johtui sinänsä vain siitä, että kuvien piirtämiseen jokaista animaation vaihetta varten kuluu runsaasti aikaa, ja tärkeämpää oli saada ohjelman rakenne ja koodi parempaan kuntoon kuin piirtää lisää kuvia. Peliin on kuitenkin helppo lisätä hahmoja, jos kuvat ovat valmiina tai niitä piirtää lisää, joten tämä ero ei varsinaisesti liity ohjelmointiin mitenkään.

12. Toteutunut työjärjestys ja aikataulu

Projekti toteutettiin pääpiirteittäin seuraavassa järjestyksessä:

Taustalogiikan toteutus (n. 9.3. - 25.3.)
Kartta ja ruudut
Hahmot ja niiden liikkuminen
Toiminnot
Vuorojen toiminta
Asetustiedostojen lukeminen (ConfigFileReader)

Käyttöliittymän toteutus (n. 25.3. - 15.4.)
Joidenkin ruutujen ja hahmojen piirtäminen käyttöliittymän rakentamista varten
Pygamen opettelua (esim. klikatun ruudun tunnistaminen, hahmojen kävely)
Vähitellen pelin kasaaminen pääsilmukkaan
Useiden hahmojen ja ruutujen piirtämistä
Päävalikko ilman muita valikoita

Tekoälyn toteutus (n. 10.4. - 20.4.)
Kohteen valinta pelkästään etäisyyden perusteella
Toiminnon valinta

Ohjelman rakenteen uudelleen toteutus (n. 20.4. - 10.5.)
Ohjelmatilat (State, StateManager)
Tapahtumankäsittelijät (EventHandler)
Pääsilmukan kasaaminen näiden avulla
Itse pelin rakentaminen uudelleen omaan tilaansa
Valikkojen rakentaminen (Menu, MenuOption)
Äänien lisäys, musiikit ja ääniefektit
Tekoälyn parantamista ja mukautuvuuden lisäämistä
Parantelua ja ominaisuuksien lisäämistä

Koko projektin aikana tehtiin testaamista manuaalisesti ja melko alusta asti myös osittain yksikkötestien avulla. Kuten edellä mainitsin, suurin muutos aikataulullisesti suunnitelmaan tuli ohjelman rakenteen uudelleen toteuttamisesta, ja siihen kului ajallisesti suunnilleen yhtä

paljon aikaa kuin sitä edeltäviin vaiheisiin, vaikka se tehtiin päivämäärällisesti pienemmällä välillä.

Joitakin pienempiä poikkeamia oli myös. Esimerkiksi vuorojen toiminnan rakentamiseen ei kulunut lähellekään suunniteltua kuutta tuntia, ja tämä aika siirtyikin suurelta osin asetustiedostojen lukemisen toteuttamiseen. Suurimman osan asioista toteuttamiseen kului hieman enemmän aikaa kuin suunnitelmissa oli, mutta suuruusluokat olivat jossain määrin oikeat.

13. Arvio lopputuloksesta

Kokonaisuutena olen ohjelmaan tyytyväinen. Sillä on mielestäni selkeä ja looginen perusrakenne, joka soveltuu hyvin tähän ohjelmaan. Ohjelmassa on eri osia, kuten esimerkiksi tilanhallinta, tilat, tapahtumankäsittelijät, pelin taustalogiikka ja pelin näkymät, ja nämä osaset on eroteltu omiksi komponenteikseen. Moni osa koodista soveltuisi varmasti hyvin käytettäväksi pohjana muissakin projekteissa. Luokkarakenne hyödyntää perintää ja myös abstrakteja luokkia, esimerkiksi tilojen ja valikoiden kanssa.

Käyttämäni algoritmit toimivat sopivan tehokkaasti niin, että niiden toimintaa ei pelin taustalla huomaa, ja ne sopeutuvat pelin eli tilanteisiin joko automaattisesti tai parametrien avulla. Mielestäni ne ovat myös sopivat algoritmit vaadittuihin tehtäviin, vaikka muitakin algoritmeja olisi valittavissa, sillä ne ovat riittävän nopeita ja kuitenkin helppoja ymmärtää. Jotkin muut algoritmit saattaisivat olla nopeampia, mutta samalla vaikeampia, ja algoritmien nopeus ei tässä tapauksessa muodostu ongelmaksi, koska pelin karttaruudukko on suhteellisen pieni.

Peli on pitkälle konfiguroitavissa asetustiedostojen avulla, kaikki kartat ja hahmot voidaan määrittää itse ja niitä voi tehdä haluamansa määrän. Pelin teemaa olisikin helppo muuttaa vain piirtämällä uusia kuvia.

Asetustenlukija on myös toteutettu niin, että se osaa käsitellä tuntemattomatkin asetukset. Jos esimerkiksi lisättäisiin peliin ominaisuuksia, voitaisiin vain lisätä niitä asetustiedostoon ja sen jälkeen suoraan rakentajametodeihin, koska itse tiedostot lukeva metodi osaa käsitellä kaikki oikein muotoillut asetukset tiedostoista. Lukija myös ilmoittaa virheistä melko tarkasti.

Pelistä puuttuu animaatiot toiminnoille ja hahmon kuolemiselle, johtuen grafiikoiden piirtämiseen kuluvasta ajasta, mutta nämä olisi melko pienellä vaivalla toteutettavissa peliin koodin puolesta, jos jatkaisin sen kehitystä. Kävelyanimaatiot toimivat hyvin, ja muut animaatiot voitaisiin tehdä samalla periaatteella sillä erotuksella, että toiminnon ja kuoleman animaatio toistettaisiin vain kerran. Tämä ehkä hieman latistaa muuten mukavaa tunnelmaa pelissä.

Tekoäly pelaa sujuvasti peliä ja mukautuu erilaisiin tilanteisiin ja toimintoihin, mutta jos jatkaisin kehitystä, edistäisin sitä vielä pidemmälle pois satunnaisuudesta. Samoin korjaisin sen tilanteen, jossa tekoäly ei tällä hetkellä tiedä mitä tehdä, eli tilanteen, jossa yhteenkään kohteeseen ei ole polkua. Jos kaikki mahdolliset reitit tukitaan, odottaa tekoäly reitin vapautumista, ja lähtee jälleen liikkumaan vuorollaan, jos reitti on vapaa. Tämä tilanne aiheutuu siitä, että tekoälyn halutaan myös kiertävän esteet, jolloin käytetään lyhimmän polun etsivää algoritmia, joka kiertää hahmot ja ruudut, joihin ei voi kävellä. Polkua ei siis löydy, jos reitit on tukittu hahmoilla, ja ilman reittiä tekoäly ei osaa liikkua.

Mielestäni luokkarakenne, johon lopulta päädyin, on toimiva, ja se soveltuu laajennusten tekemiseen. Esimerkiksi uusia valikoita voitaisiin lisätä hyvinkin helposti, ja valikotkin sopeutuvat annettujen vaihtoehtojen määrään ja esimerkiksi luettujen karttojen määrään. Komponentit erotellaan tehtävien mukaan toisistaan, ja ne toimivat hyvin yhdessä.

Lopputulokseen päädyttiin osittain kantapään kautta, koska suunnitelmassa ei oltu otettu huomioon tarpeeksi käyttöliittymään liittyviä asioita. Ja koska ohjelma on kuitenkin graafisen käyttöliittymän kautta käytettävä, kaikki sen osiot liittyvät jotain kautta käyttöliittymän mallintamiseen käytettyihin luokkiinkin. Aluksi toteutin lähes kaiken pääsilmukkaan, mikä oli erittäin huono ja tehoton ratkaisu. Koodia oli vaikea kirjoittaa lisää ja ymmärtää sen rakennetta jopa itse. Tähän vaiheeseen mennessä kuitenkin olin oppinut perusteet Pygamen avulla käyttöliittymän piirtämisestä, joten oli helpompi suunnitella rakenteeseen vaadittavat muutokset. Ohjelman kehitys oli muutenkin hyvin iteratiivinen prosessi, sillä osasia toteutettiin ja niitä paranneltiin, mietittiin ja toteutettiin uudelleen, kun asioita opittiin lisää. Tämä johtui varmasti osittain siitä, että pelissä tarvittavia käyttöliittymätaitoja ei suuremmin opetettu luennoilla, mutta melko helposti ne lopulta oppi itsekin.

Lopputulokseen olen kokonaisuudessaan hyvin tyytyväinen. Onnistuin mielestäni kasaamaan loogisen rakenteen ohjelmalle, ja käyttämään hyvin hyödyksi myös Pygamen tarjoamia ominaisuuksia, jopa melko edistyneitä ominaisuuksia, kuten sen tapahtumapinoa itse määrittelemilläni tapahtumilla. Ohjelma on laajennettavissa ja muokattavissa Pelistä puuttuu pari palasta, kuten tietyt animaatiot, mutta ne olisi helppo lisätä myöhemmin, ja mielestäni mitään kriittistä ei puutu. Animaatioiden puuttuminen johtui lähinnä piirtämisen vaatimasta ajasta, sillä tein kaikki grafiikat itse. Pelissä on silti hyvä tunnelma, ja sitä on ollut itsekin hauska pelata testatessa.

14. Viitteet

Pygamen versio, joka toimii 64bit Pythonin (3.4) kanssa saatavilla täältä .whl muodossa
http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame

Edellisten kierrosten harjoitustehtävät
esim. Robots -tehtävä ja shakkitehtävä

Pythonin dokumentaatio
Erittäin runsaasti Python 3:n dokumentaation lukemista
https://docs.python.org/3/

Pygamen dokumentaatio ja sivusto käyttöliittymän toteuttamista varten
Runsaasti myös Pygamen dokumentaation lukemista
http://www.pygame.org/docs/
http://pygame.org/news.html
http://www.pygame.org/wiki/tutorials

Pygamea opetellessa useisiin vanhoihin Pygame-peleihin ja ohjelmiin tutustumista
http://www.pygame.org/tags/strategy
http://www.pygame.org/tags/turnbased
http://www.pygame.org/project-IsoCraft-2346-.html
http://pygame.org/project-Port+Tales-2904-.html
http://blog.vim.pl/2011/11/gra-izometryczna-cz-1-wstep/

http://pygame.org/project-EzMeNu-855-.html
http://pygame.org/project-rgm-961-.html
http://pygame.org/project-AngrySnakes-2197-.html
http://pygame.org/project-menu_key-2278-.html
http://pygame.org/project-MenuClass-1260-.html
http://pygame.org/tags/menu

Gamedevelopment tuts+ sivusto, apua isometrisen pelin suunnitteluun
http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-developers--gamedev-6511
http://gamedevelopment.tutsplus.com/tutorials/cheap-and-easy-isometric-levels--gamedev-6282
http://clintbellanger.net/articles/isometric_math/

CSE-A1141 Tietorakenteet ja algoritmit -kurssin luentomateriaalit, erityisesti algoritmit-kohdassa mainittujen algoritmien suunnittelussa apua
https://noppa.aalto.fi/noppa/kurssi/cse-a1141/luennot/

PythonCentral sivusto
http://www.pythoncentral.io/how-to-sort-python-dictionaries-by-key-or-value/

StackOverflow sivusto
Runsaasti vanhojen pythoniin liittyvien kysymysten selailua ja etsimistä
http://stackoverflow.com/
http://gamedev.stackexchange.com

stupidpythonideas.blogspot.com
http://stupidpythonideas.blogspot.fi/2013/11/does-python-pass-by-value-or-by.html

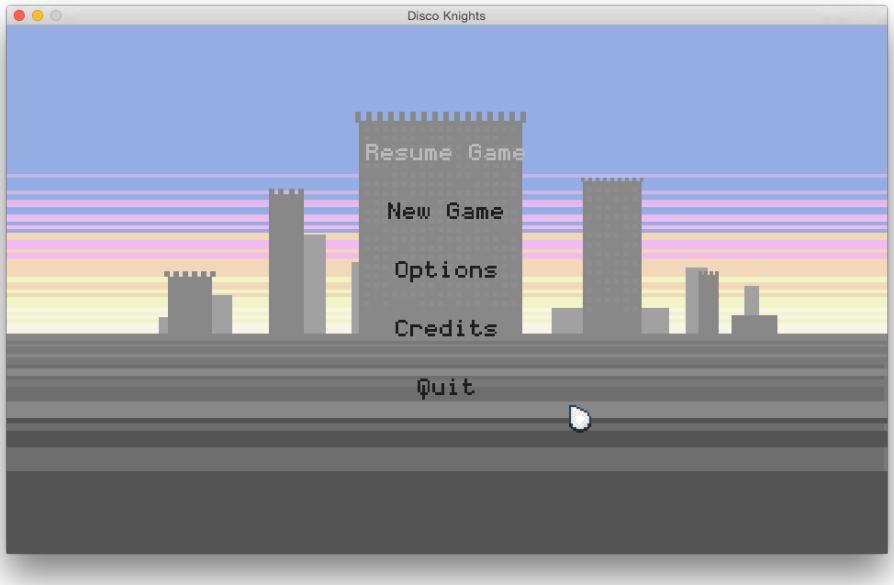
nerdparadise.com
http://nerdparadise.com/tech/python/pygame/tipsandtricks/

musiikkeja ja ääniefektejä on saatu seuraavista lähteistä
http://www.playonloop.com/2014-music-loops/yellow-copter/
http://www.allmusiclibrary.com/free_sound_effects.php
http://www.freesound.org/people/LittleRobotSoundFactory/sounds/270329/
http://www.freesound.org/people/Erdie/sounds/27858/
http://www.freesound.org/people/qubodup/sounds/185478/
http://www.freesound.org/people/joe93barlow/sounds/105986/
http://www.freesound.org/people/spookymodem/sounds/249810/
http://www.freesound.org/people/RSilveira_88/sounds/216196/
https://soundcloud.com/plrusek-chan/struck-by-the-rain-n163

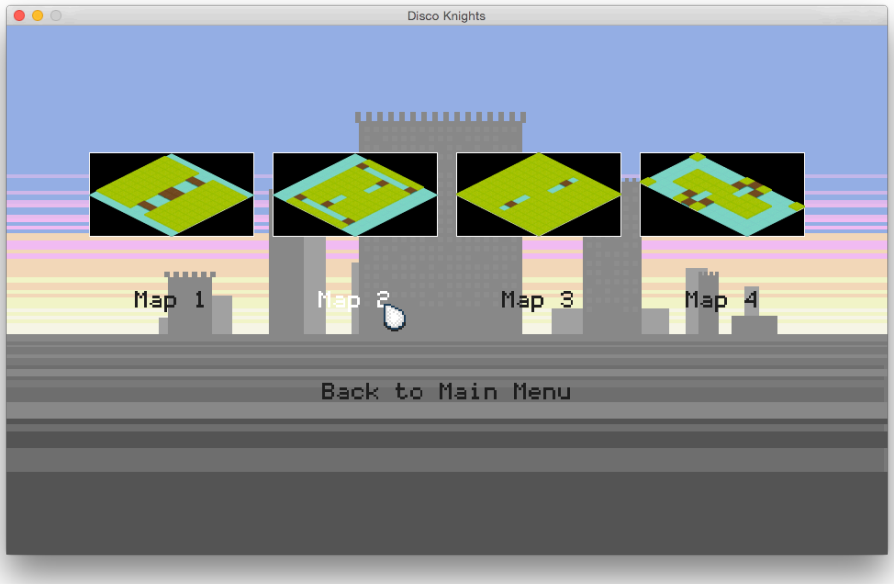
15. Liitteet

Muutamia kuvia ohjelmasta käytön aikana:

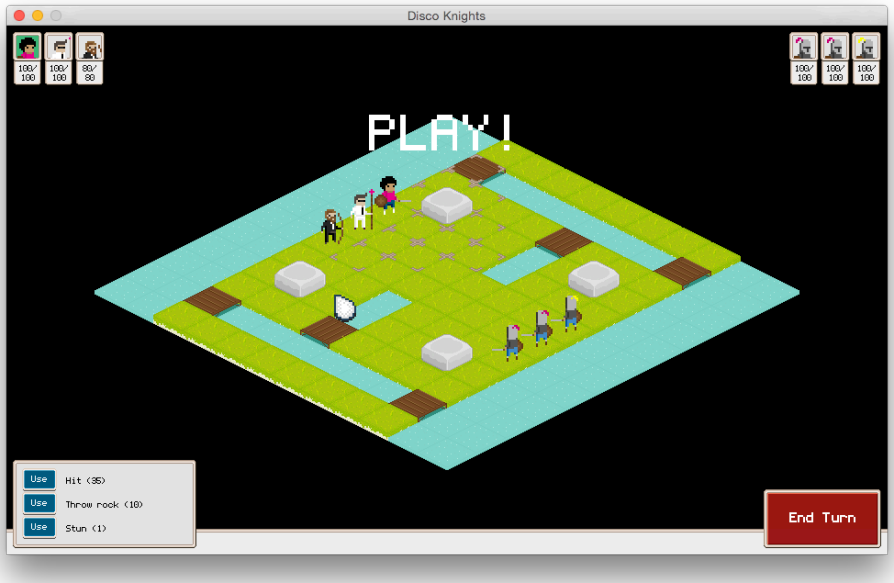
Päävalikko



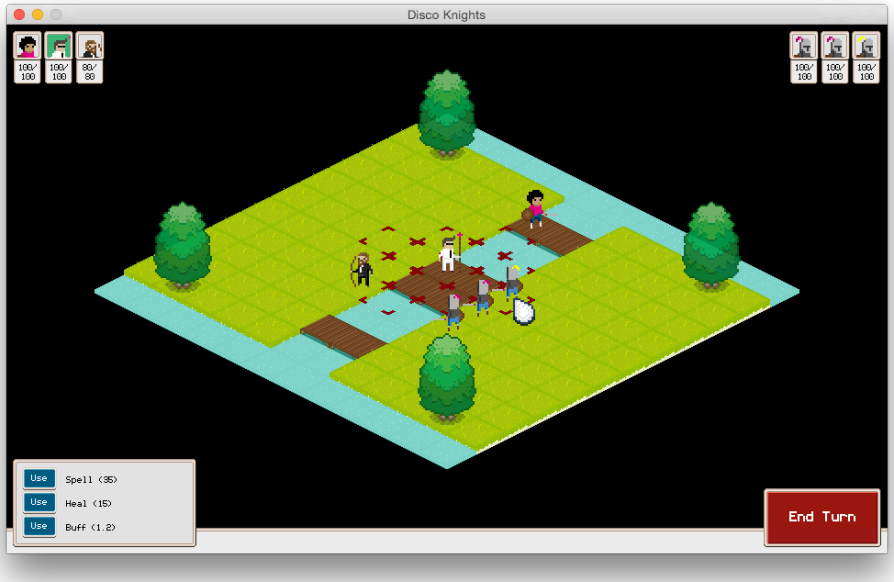
Kartan valinta



Pelin alku



Toiminnon kantaman esitys sen valinnan jälkeen



Toinen kartta, vuoro on tullut pelaajan hahmolle

Voittajan näyttäminen pelin loputtua