Ohjelmointistudio 2 | CS-C2120

*Projektityön dokumentti*

**Tower Defense**

*Tornipuolustuspeli*

Jussi Nevavuori | 709097

*Tietotekniikka, syksy 2018*

15.4.2019

Sisällysluettelo

[1. Yleiskuvaus 3](#_Toc6320528)

[2. Käyttöohje 4](#_Toc6320529)

[3. Ohjelman rakenne 7](#_Toc6320530)

[4. Algoritmit 17](#_Toc6320531)

[5. Tietorakenteet 22](#_Toc6320532)

[6. Tiedostot ja verkossa oleva tieto 23](#_Toc6320533)

[7. Testaus 25](#_Toc6320534)

[8. Ohjelman tunnetut puutteet ja viat 26](#_Toc6320535)

[9. Kolme parasta kohtaa 27](#_Toc6320536)

[10. Kolme heikointa kohtaa 28](#_Toc6320537)

[11. Suunnitelman ja aikataulun toteutuminen 29](#_Toc6320538)

[12. Kokonaisarvio lopputuloksesta 30](#_Toc6320539)

[13. Viitteet 31](#_Toc6320540)

[14. Liitteet 31](#_Toc6320541)

[15. Näyttökaappauksia 32](#_Toc6320542)

# Yleiskuvaus

Kyseinen projektityö on nimeltään Tower Defense, eli tornipuolustuspeli. Peliin on lisätty myös muita ominaisuuksia, minkä takia uskon, että työ on tehty hyvin haastavantasoisena.

Itse peli koostuu reitistä, jota pitkin viholliset kulkevat 'aalloissa'. Pelissä on erilaisia 'torneja', joita pelaaja voi ostaa reitin varrelle ja kehittää voimakkaammiksi rahaa vastaan. Torneilla pelaaja yrittää estää vihollisten kulun reitin loppuun, sillä loppuun päässeet viholliset vievät pelaajalta osumapisteitä, HP:ta.

Pelin ohella on myös olemassa 'level editor', jossa pelaaja pääsee itse luomaan omia tasojaan, nimeämään niitä ja tallentamaan niitä myöhemmin pelattaviksi.

Ohjelmassa on useita eri tiloja ja erilaisia valikoita ja sovellus on hyvin graafinen.

# Käyttöohje

**Käynnistäminen**

Pelin käynnistäminen tapahtuu käynnistämällä tiedoston src/gui/Main.scala scala-applikaationa. Käynnistettyään tiedoston, käyttäjälle aukeaa ikkuna, jossa ensin näkyy animoituna pelin logo. Tämän logon voi ohittaa painamalla välilyöntiä tai sen voi antaa haihtua itsestään. Tämän jälkeen pelaaja saapuu päävalikkoon. Jokaisen näkymän oikeassa yläreunassa on nuotti-ikoni, josta pelaaja voi laittaa musiikin pois päältä. Ikkunan kokoa voi vapaasti säätää ja painamalla F11 missä tahansa näkymässä saa ikkunan koko näytölle. Päävalikon 'Exit' -napista saa koko sovelluksen suljettua.

**Pelinäkymä**

Päävalikosta käyttäjä voi ylimpänä valita vaihtoehdon "New game", jolloin käyttäjälle aukeaa itse pelinäkymä ja pelin ainoa valmiiksilaadittu kenttä. Pelinäkymässä hiiren viemällä ruudun ylälaitaan aukeaa valikko, josta voi tallentaa nykyisen pelitilanteen myöhemmin (Save) jatkettavaksi, näyttää mahdolliset näppäinkomennot (Controls), nähdä GUI:n FPS-nopeuden (Show FPS), käyttää huijauskomentoja avaten kaikki tornit ja antaen pelaajalle 1000 HP ja 100000 $ (Godmode) tai palata päävalikkoon (Main Menu tai 'ESC'-näppäin). Pelin alalaidassa on vasemmalla pelaajan osumapisteet (HP) ja raha ($). Näistä oikealle on kolme tornia, joista kaksi on alussa lukittuja ja tummennettuja. Tornien alla on kunkin tornin hinta. Oikeassa alareunassa on nykyisen vihollisaallon numero, nopeutusnappi ja seuraavan aallon nappi.

**Pelaaminen: tornit**

Klikkaamalla torneja, jos pelaajalla on tarpeeksi rahaa, tulee pelaaja ostaneeksi kyseisen tornin. Torneja voi myös ostaa numeronäppäimillä 1, 2 ja 3. Pelaaja voi nyt viedä hiirellä tornin haluamaansa kohtaan ja klikata, jolloin jos kohta on sallittu, torni jää pelaajan valitsemaan kohtaan. Viedessään tornia pelaaja näkee tornin näköetäisyyden ympyränä. Tornia klikkaamalla voi valita tornin, jolloin taas näkee sen näköetäisyyden, ja jos mahdollista, sen päivityshinnan. Päivityshintaa klikkaamalla ja rahan siihen riittäessä pelaaja voi päivittää

tornin, jolloin tornin ulkomuoto ja näkösäde (ja tornin ominaisuudet) muuttuvat ja paranevat.

Tornit toimivat pelissä täysin autonomisesti ja niitä voi ostaa millä tahansa pelin hetkellä.

**Pelaaminen: viholliset**

Klikkaamalla seuraavan aallon nappia oikeasta alakulmasta tai painamalla välilyöntiä tulee pelaaja kutsuneeksi seuraavan vihollisaallon, jolloin reitin toisesta päästä alkaa ilmestymään vihollisia. Viholliset ja tornit toimivat autonomisesti, eikä pelaaja voi tehdä muuta kuin päivittää olemassa olevia torneja tai ostaa uusia. Jokaisen vihollisen kuoltua tai niiden päästyä reitin loppuun voi pelaaja taas kutsua seuraavan aallon. Aaltojen etenemistä voi nopeuttaa painamalla nopeutusnappia tai välilyöntiä pohjaan, jolloin peli etenee nopeutetusti. Tornien tappaessa vihollisia pelaaja saa rahaa, mutta viholliset voivat jakaantua useammiksi pienemmiksi vihollisiksi. Suuremmilla aalloilla viholliset ovat vahvempia.

**Pelaaminen: muuta**

Pelissä näkyvät koristeet, kuten puut ja puskat ovat täysin esteettisiä, eikä ne vaikuta peliin mitenkään. Torneja on mahdollista ostaa koristeiden päälle. Muutaman aallon välein pelissä myös vaihtuu vuodenaika, jolloin koko pelin grafiikat vaihtuvat tasaisesti. Tämäkään ei vaikuta peliin muuta kuin esteettisesti.

**Pelaaminen: tallentaminen ja jatkaminen**

Peli tallentuu joko manuaalisesti pelinäkymän ylävalikon 'Save' -napista tai automaattisesti poistuttaessa pelinäkymästä päävalikkoon. (Nykyinen peli häviää jos sovellus suljetaan suoraan pelinäkymästä.) Tallentaminen palauttaa pelitilanteen (viholliset, tornit, rahat ja osumapisteet) nykyisen aallon alkuun. Viimeiseksi tallennetun pelin jatkaminen on mahdollista päävalikon 'Continue' -napista.

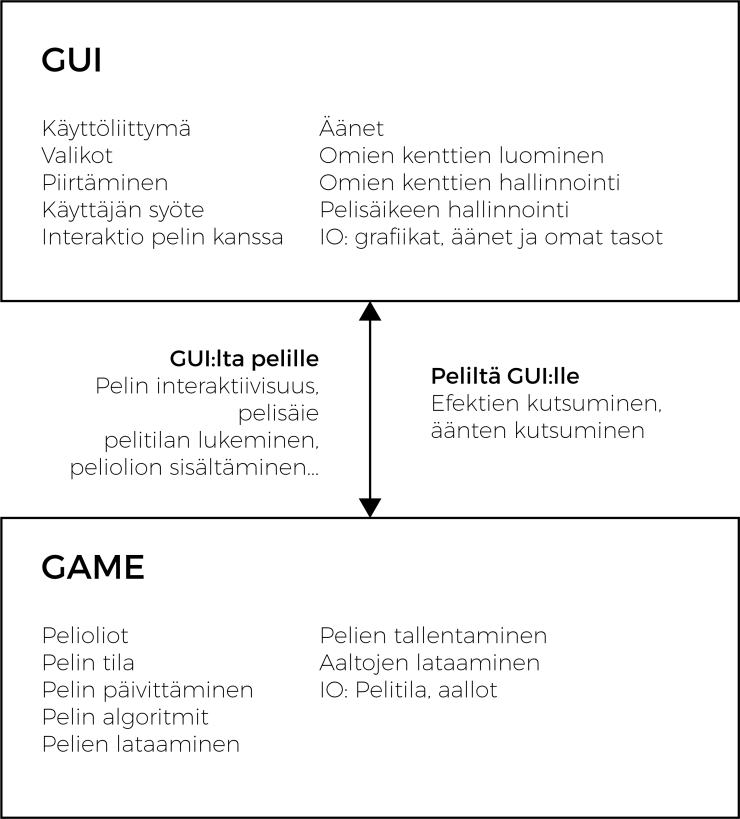
**Päävalikko: Load Level**

Päävalikon 'Load level' -nappi vie käyttäjän seuraavaan valikkoon. Valikossa on listattuna kaikki käyttäjän itsensä luomat kentät, joissa voi aloittaa uuden pelin painamalla niiden nimeä, jolloin päätyy suoraan pelinäkymään, kyseiseen kenttään tai niitä voi poistaa niiden vieressä sijaitsevasta punaisesta rastista. Jos käyttäjällä ei ole yhtään omia tasoja, lukee valikossa 'No custom levels'. Valikon alareunasta on mahdollista palata takaisin päävalikkoon.

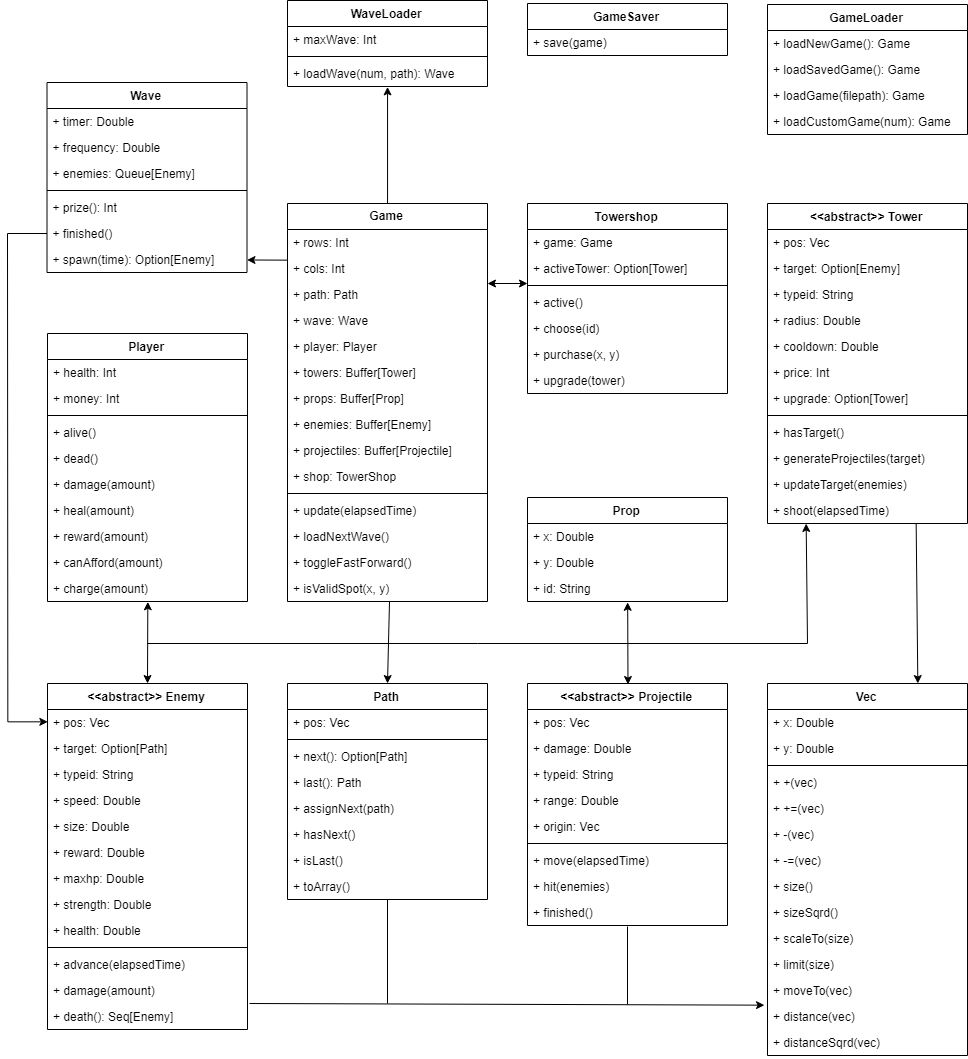
**Level Editor**

Käyttäjä pääsee Level Editoriin painamalla päävalikon 'Level Editor' -nappia, jolloin aukeaa Level Editor näkymä. Näkymä on samannäköinen, kuin pelinäkymä, mutta kenttä on tyhjä ja alalaidassa on eri elementtejä. Klikkaamalla mihin tahansa kentän reunalle voi pelaaja aloittaa luomaan uutta reittiä tästä kohtaa. Nyt joko hiirellä, tai mieluummin nuolinäppäimillä pelaaja voi ohjata reittiä. Reitti voi kulkea suoraan, kääntyä tai jopa hypätä itsensä yli. ***HUOM:*** *Jotta reitti olisi tallennettavissa ja toimiva, on reitin viimeisen palan oltava reunan vieressä.*

# Ohjelman rakenne

Kuten alla olevassa kaaviossa näkyy, ohjelma on jaettu raa'asti GUI:hin eli käyttöliittymään ja itse peliin, jotka sijaitsevat omissa packageissaan projektin lähdekoodissa. Itse peli sisältää pelioliot ja nykyisen pelin tilan ja huolehtii pelin päivittämisestä ja algoritmeista. Game packagessa on myös tarpeelliset IO-objektit pelien tallentamiseen, lataamiseen ja luomiseen. Pelistä GUI:hin viittauksia on yritetty minimoida, mutta hyvin yksinkertaisia viittauksia pelistä löytyy kuitenkin käyttö-liittymään. Viittaukset ovat kuitenkin pelkästään efektien luomista (graafisia efektejä varten, kuten vihollisen kuollessa tai päivittäessä tornin) tai äänten toistamista varten (esimerkiksi tornin ampuessa). GUI taas pääsee pelipakkauksen tietoihin käsiksi paljon. Se pääosin lukee niitä, esimerkiksi piirtääkseen pelin oikein, mutta myös muokkaa pelin tietoja esimerkiksi käyttäjän syötteen pyynnöstä, mahdollistaen interaktiivisuuden.

Seuraavaksi löytyy molempien pakkauksien UML-diagrammit, joiden molempien jälkeen on avattu pakkauksissa käytettyjä luokkia.



(UML-Kaavio: Game package)

**Game-package:**

Pelipakkauksen keskiössä on itse peliluokka, **Game**, joka kuvaa peliä. Se sisältää pelitilanteen ja kaikki pelin oliot, kuten viholliset, pelaajan, polun jne. Peliolioilla on myös tärkeimpänä metodina *update(elapsedTime)*, jolla peli osaa päivittää koko pelitilanteen kuluneen ajan verran, esimerkiksi päivittämällä kaikki pelin oliot ja poistamalla kuolleet viholliset.

Peliolioihin lukeutuu pelaaja, eli **Player**-olio, joka sisältää pelaajan rahat ja osumapisteet ja metodeja näiden käsittelyyn. Pelaajan lisäksi pelissä on aina lista **Prop**-olioita, jotka kuvastavat staattisia koristeita pelissä.

Monessa luokassa sijaintia, liikettä ja nopeuksia kuvataan vektoreilla, minkä takia pelissä on myös yksinkertainen kaksiulotteinen vektoriluokka **Vec**, joka sisältää x- ja y-arvot ja erilaisia perus vektorifunktioita, joita on pelissä käytetty.

Peli sisältää myös polkuolion, **Path**. Polkuoliot ovat rakennettuja niin, että jokaisesta polkuoliosta on myös viittaus seuraavaan polkuun, jos sellainen on olemassa. Tämän rakenteen avulla riittää, että peliolio säilöö vain ensimmäisen polun, sillä tästä polusta muodostuu ketju, joka sisältää kaikki muut polut. Jokaisella polulla on myös oma sijainti paikkavektorina.

Peli sisältää myös vihollisia, **Enemy***,* joka on ensimmäinen abstrakti luokka. Vihollisilla on sijainti paikkavektorissa, kohde *target*, johon on tallennettu se polku, jota kohti vihollinen parhaillaan kulkee. Tässä polkujen ketjurakenne mahdollistaa vihollisten helpon liikkumisen polkua pitkin. Vihollisilla on myös lukuisia ominaisuuksia, kuten nopeus, koko tai osumapisteet. Vihollisen tärkeimpiä metodeja ovat *advance(elapsedTime)*, jonka avulla vihollinen liikkuu, *damage(amount)*, jolla vihollista vahingoitetaan ja *death(): Seq[Enemy]*, joka palauttaa listan vihollisista, jotka syntyvät tämän vihollisen tilalle vihollisen kuollessa. Vihollisluokkaa perii moni konkreettinen vihollisluokka, jotka eroavat toisistaan vain ominaisuuksillaan, muttei toiminnallisuudeltaan. Näin on saatu toteutettua erilaisia ja eri vahvuisia vihollisia.

Torniluokka, **Tower***,* on seuraava abstraktiluokka. Torneilla on myös sijainti paikkavektorissa, kohde *target*, joka on se vihollinen, jota päin torni parhaillaan tähtää ja *upgrade*, joka on se torni, joksi tämän tornin voi päivittää. Myös torneilla on useita eri ominaisuuksia, joita torniluokkaa perivät konkreettiset tornityyppiluokat täydentävät eri tavalla. Näin on saatu toteutettua myös useita eri tornityyppejä. Tornien tärkeimmät metodit ovat *updateTarget(enemies)*, jonka avulla vihollinen yrittää koko ajan tarvittaessa löytää uuden kohteen, *shoot(elapsedTime)*, jonka avulla vihollinen yrittää *cooldown*-ominaisuuden määrittelemän väliajan välein ampua ja *generateProjectiles(target)*, joka palauttaa listan projektiileista, jotka tämä torni synnyttää ampuessaan.

Viimeinen abstrakti luokka on projektiililuokka, **Projectile***,* joka kuvaa vihollisten ampumia projektiileja. Kaikilla projektiileilla on sijainti, *pos* ja ominaisuuksia, kuten vahinko, *damage* tai kantama, *range*. Projektiilien tärkeät metodit ovat *move(elapsedTime)*, jonka avulla projektiili liikkuu ja *hit(enemies),* jonka avulla projektiili yrittää osua vihollisiin. Konkreettiset, projektiililuokkaa perivät projektiilityyppiluokat *(bullet, missile ja boomerang)* eroavat toisistaan hieman ominaisuuksiltaan, mutta etenkin metodeiltaan *move()* ja *hit()*. Luodit, *bullet*, liikkuvat suoraviivaisesti vakionopeudella kantamansa verran ja katoavat, elleivät osu viholliseen. Bumerangit*, boomerang*, liikkuvat suoraviivaisesti eteenpäin, kokoajan kiihtyen takaisinpäin ja ne kääntävät suuntansa ja palaavat takaisin tornille, bumerangin tavoin. Bumerangit taas voivat osua useaan viholliseen ja samaan viholliseen jopa useamman kerran ja katoavat vasta palattuaan tornille. Ohjukset, *missile*, taas kiihdyttävät saavuttaen huippunopeutensa ja kulkeutuvat kauas, ohjautuen automaattisesti päin niiden kohdetta (jos kohde on vielä elossa). Ohjuksen osuessa ohjus vahingoittaa myös muita lähellä olevia vihollisia.

Tornien ostamista ja pelikentälle asettamista varten on olemassa luokka **Towershop**, tornikauppa. Tornikaupassa on tärkeä luokka käyttöliittymän ja pelin interaktiota varten. Tornikaupassa on viittaus siihen peliin, johon se kuuluu muuttujassa *game*. Tornikaupassa on myös nykyinen aktiivinen torni, *activeTower*, joka sisältää (jos sisältää) sen tornin, jota käyttäjä on ostamassa. Tornikaupan metodilla *choose(id)* voi valita aktiiviseksi torniksi id:n määräämän tornityypin. Kun pelaaja päättää sijainnin ostamalleen tornille, metodi *purchase(x, y)* lisää tornin oikealle paikalleen ja lisää sen peliin. Metodilla *upgrade(tower)* tornikauppa osaa päivittää valitun tornin.

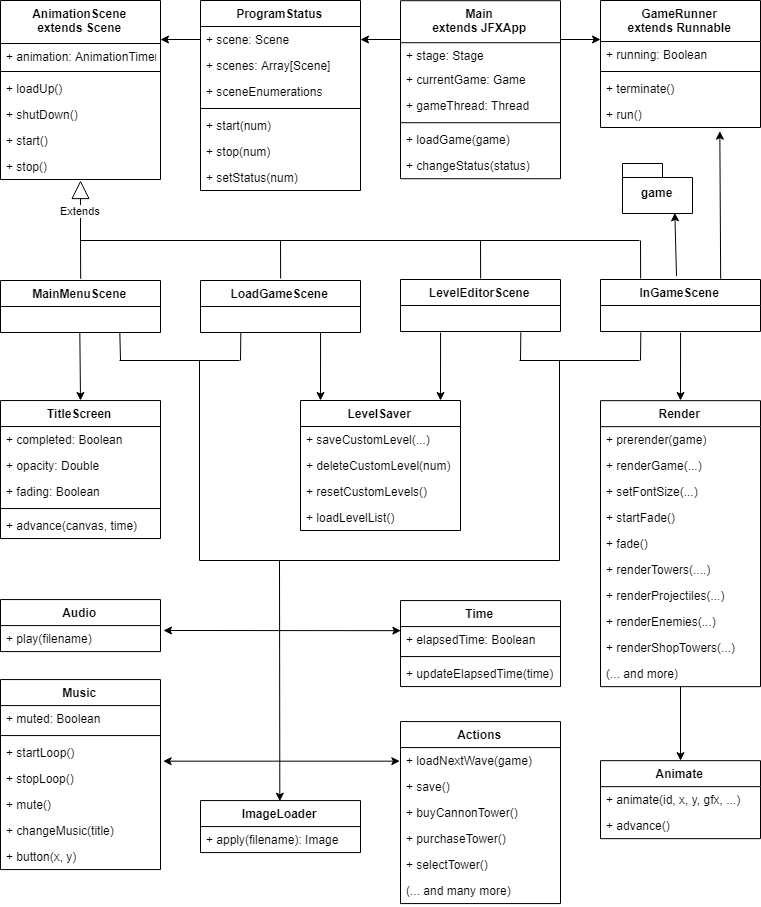
Vihollisaaltoja kuvaava **Wave** luokka taas sisältää jonollisen vihollisia, ajastimen ja tietyn taajuuden, jonka avulla metodia *spawn(time)* kutsumalla saadaan tasaisin väliajoin otettua jonossa ensimmäinen vihollinen ja lisättyä se peliin. Metodi *prize()* palauttaa kyseisestä aallosta saatavan rahapalkkion, mutta vain kerran.

Paketissa on myös kolme tärkeää IO-objektia. **WaveLoader** objekti sisältää tiedon aaltojen lukumäärästä muuttujassa *maxWave* ja metodin *loadWave(num, path): Wave*, jolla se lataa tiedostosta pyydetyn aallon, luo sen viholliset halutulle polulle ja palauttaa luodun aallon.

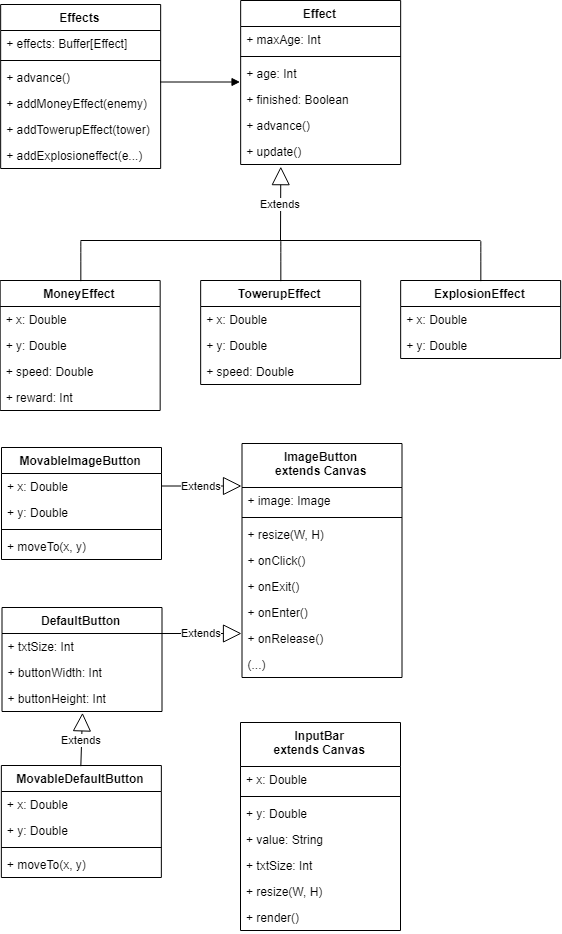
**GameSaver** objektilla on vain yksi metodi, *save(game)* jolla se osaa tallentaa sille annetun pelin tiedostoon, joka sisältää viimeisimmäksi tallennetun pelin.

**GameLoader** objekti sisältää kolme metodia, joista *loadGame(filepath): Game* lataa ja luo sille annetussa tiedostopolussa sijaitsevasta tiedostosta pelin ja palauttaa sen. GameLoaderin *loadNewGame()* ja *loadSavedGame()* metodit luovat uuden tyhjän pelin vakiotiedostosta tai lataavat tiedostosta viimeksi tallennetun pelin ja palauttavat sen. Metodi *loadCustomGame(num)* lataa halutun level editorissa rakennetun oman kentän ja palauttaa sen.

Yhteenvetona suurin osa pelipakkauksen luokista ovat hyvin luonnollisia olio-ohjelmoinnin tapaisia abstrahointeja pelin liikkuvista osista ja niiden välisestä interaktiosta. Ainoat, jotka poikkeavat tästä luonnollisesta kokonaisuudesta, ovat TowerShop, joka on rakennettu GUI:lle käyttöliittymäksi tornien ostamista ja päivittämistä varten ja kolme eri IO-objektia, jotka osaavat ladata pelin tietoja tiedostosta tai tallentaa pelitilanteita tiedostoon.



(UML-Kaavio: GUI package 1)



(UML-Kaavio: GUI package 2)

**GUI package:**

GUI:n keskellä on **Main-**objekti, joka perii ScalaFX-kirjaston JFXApp luokkaa ja josta koko sovellus käynnistetään. Koko GUI on rakennettu käyttämällä ScalaFX-kirjastoa. Main osaa rakentaa ikkunan ja hakea ikkunaan oikean näkymän, eli scenen. Main pitää sisällään myös tämän hetkisen pelin *currentGame* ja peliä ajavan rinnakkaisen säikeen *gameThread*. Main-objektilla on myös metodeja näkymän vaihtamiseen *changeStatus(status)* ja uuden pelin lataamiseen nykyiseksi peliksi *loadGame(game)*. Main objekti tekee paljon yhteistyötä **ProgramStatus**-objektin kanssa, joka pitää sisällään listan kaikista näkymistä ja sisältää metodeja näkymien vaihtamiseksi ja niiden hallitsemiseen.

Jokainen näkymä perii luokkaa **AnimationScene**, joka puolestaan perii ScalaFX:n Scene-luokkaa ja implementoi muutamia käteviä metodeja, jotka helpottavat näkymien luomista ja parantavat niiden yhteensopivuutta muun käyttöliittymän kanssa. AnimationScenen tärkeimpiä ominaisuuksia ovat se, että niillä on AnimationTimer -luokan animaatio, jossa jokainen näkymä pystyy animoimaan itseään tai tekemään muita toimenpiteitä 60 kertaa sekunnissa. AnimationScene -näkymillä on myös metodeja, jotka ajetaan aina näkymän avattaessa tai suljettaessa ja metodeja, joilla animaation voi käynnistää ja pysäyttää. Näkymät sisältävät valikoita, painikkeita ja 'canvaksia', joiden avulla näkymät saadaan piirrettyä.

Käyttöliittymän keskiössä ovat neljä erilaista näkymää, joiden sisäistä toiminnallisuutta ei ole UML-kaavioon avattu. Jokainen niistä perii AnimationScene-luokkaa. Ensimmäinen näkymistä on päävalikkonäkymä, eli **MainMenuScene**-objekti. MainMenuScene pitää sisällään valikon, josta pääsee siirtymään muihin näkymiin. Ohjelman avattaessa MainMenuScene näkyy ensimmäisenä näkymänä. Siksi MainMenuScene toimii yhdessä **TitleScreen**-objektin kanssa, etenkin kutsuen sen *advance()* metodia animaatiossaan. MainMenuScene animoi siten kerran ohjelman avattaessa otsikko-animaation, jossa näkyy pelin logo.

Seuraava näkymä on **LoadGameScene**, joka pyytää **LevelSaver**-objektilta aina avattaessa listan kaikista käyttäjän luomista kentistä ja rakentaa rullattavan valikon, jossa on vaihtoehdot avata uusi peli jokaisessa kentässä tai poistaa se kenttä.

Hieman monimutkaisempi näkymä on **LevelEditorScene**, joka sisältää Level Editorin. LevelEditorScene piirtää kuvan tyhjästä kentästä ja alareunan valikosta ja sisältää toiminnallisuuden, jossa hiirellä ja/tai nuolinäppäimillä voi rakentaa polkua ja hiirellä voi viedä polulle koristeita. LevelEditorScenestä on myös mahdollista tallentaa ja nimetä luomia tasojaan, mikä on toteutettu yhteistyöllä LevelSaver-objektin kanssa.

Viimeinen ja tärkein näkymä on **InGameScene**, joka on itse pelinäkymä. Pelinäkymässä osaa avattaessa luoda **GameRunner**-objektista Main-objektiin uuden *gameThread* säikeen, joka on niin kauan aktiivinen, kuin pelinäkymä on auki. Suljettaessa pelinäkymä osaa myös sulkea pelisäikeen ja pysäyttää siten pelin etenemisen. GameRunner-objekti sisältää tärkeimmän metodinsa, *run()*, jonka avulla säikeen käynnistettäessä säie alkaa päivittämään peliä silmukassa rinnakkaisesti käyttöliittymän kanssa. Rinnakkaisen säikeen päivittäessä peliä, pelinäkymä kutsuu animaatiossaan **Render**-objektia ja Render puolestaan **Animate-**objektia, jotka lukevat nykyisen pelitilanteen ja piirtävät sen avulla pelin näkymään. Render sisältää useita funktioita, jotka piirtävät staattisia grafiikoita, kun Animate puolestaan osaa piirtää animoituvia grafiikoita. Pelinäkymä sisältää myös paljon toiminnallisuuksia ja nappeja, jotka joko vaikuttavat ja kutsuvat jollain tavalla game packagen luokkia ja objekteja tai lukevat sieltä tietoa.

Kaikki käyttöliittymän näkymät käyttävät avukseen myös useita apuobjekteja, kuten **Audio**, jonka ainoalla metodilla *play(filename)* voi toistaa ääniä tai **Music**, jolla on erikoistuneet funktionaalisuudet juuri taustamusiikin toistamiseen, vaihtamiseen tai hiljentämiseen. Moni luokka käyttää myös avukseen **ImageLoader**-objektia, jonka ainoa metodi osaa ladata pyydetyn kuvan tiedostosta ja palauttaa sen. **Actions**-objekti taas pitää sisällään lukuisia eri metodeja, jotta monimutkaisiakin asioita voidaan käyttöliittymän koodissa suorittaa yksittäisillä metodikutsuilla. Jokainen animaationäkymä taas animaatiossaan päivittää koko ajan **Time**-objektin *elapsedTime* muuttujaa kutsumalla sen metodia *updateElapsedTime(time)*, jonka avulla voidaan kaikkialta käyttöliittymästä lukea koko ajan päivitysten välinen aika.

Seuraavalla sivulla UML-kaaviossa on vielä lueteltu efektit, jotka on **Effects**-objektin sisällä. Effects-objekti sisältää abstraktin luokan **Effect**, jota perii kolme eri efektiluokkaa, joista jokainen toimii hieman eri tavalla. Effect luokalla on metodit kaikkien efektien päivittämiseen kerralla ja uusien efektien lisäämiseen. Efektit ovat pääasiassa olioita, joiden ainoa syy on näkyä graafisesti pelinäkymässä, koristeina. Effects yhdessä Audion kanssa ovat GUI packagen ainoat objektit, joita game packagesta kutsutaan.

Käyttöliittymien rakentamista on auttanut useat eri itse tarkoitukseen räätälöidyt elementit, kuten **ImageButton** ja **MovableImageButton** -luokat, joiden avulla voidaan mikä tahansa kuva muuttaa toimivaksi ja interaktiiviseksi napiksi ja **DefaultButton** ja **MovableDefault-Button**, joiden avulla voidaan rakentaa juuri pelin ulkonäköön sopivia nappeja, joissa lukee jokin teksti. Näiden lisäksi on luotu **InputBar** -luokka, jota käytetään LevelEditorissa erikoistuneena tekstikenttänä.

Käyttöliittymä on siis saatu järkevästi kasaan ScalaFX:n kirjastoa apuna käyttäen. Sen rakenne on hyvin looginen: ikkuna ja sovellus on kuvattu Main-objektissa ja siitä polveutuu neljä eri näkymää, joiden välillä sovellus osaa vaihtaa ProgramStatusta avulla käyttäen. Suurin osa käyttöliittymästä on muuten vain apuobjekteja ja -luokkia, joiden avulla käyttöliittymän rakentaminen on helpottunut. Käyttöliittymästä on myös järkevästi osattu eristää GameRunner-objekti, jonka avulla on saatu oma säie pelin päivittämiseksi, mikä nopeuttaa käyttöliittymää paljon.

Ohjelman rakenteen kannalta tämä rakenne tuntuu erittäin luontevalta, eikä ohjelmalle vaikuta olevan kovin montaa muuta järkevää ja erilaista rakennetta.

# Algoritmit

Kaikki pelin algoritmit sijaitsevat pelipakkauksessa. Käyttöliittymässä ei ole mainitsemisen arvoisia algoritmeja käytetty. Tässä on mainittu joitain, näkyvimpiä algoritmeja. Pelin algoritmit eivät ole tehokkaimpia mahdollisia, eivätkä välttämättä kovin skaalautuvia, mutta pelin kappalemäärät on oletettu riittävän pieniksi, että nykyiset algoritmit selviävät lähes ongelmitta.

**Vihollisten liikkuminen**

Jokainen vihollinen tietää oman sijaintinsa, paikkavektorin **x**. Jokainen vihollinen tietää myös sen kohdepolun sijainnin, paikkavektorin **p.** Suunta, johon vihollisen tulee siis liikkua, on **b - x**. Kun viholliselle lasketaan tämä suuntavektori **p - x** ja sitä lyhennetään halutun pituiseksi liikevektoriksi **v**, saadaan vihollisen seuraava sijainti **x'** vektorisummana **x' = x + v**.

Jos huomataan että suuntavektorin **p - x** pituus on 0 (tai lähes 0), huomataan että **p = x** eli vihollinen on saapunut kohdepolkuunsa. Polut on ketjutettu, ja jokaisessa polussa on viittaus sitä seuraavaan polkuun. Tällöin vihollinen voi kohdepolultaan kysyä seuraavaa polkua, ja valita sen uudeksi kohteekseen ja jatkaa liikkumista sen suuntaan. Jos seuraavaa polkua ei ole olemassa, tiedetään että vihollinen on saapunut polun loppuun ja voidaan suorittaa tarpeelliset toimenpiteet ja poistaa vihollinen pelistä.

**Tornien kohteen hakeminen**

Torneilla on tiedossa niiden nykyinen kohde. Joka päivityksellä torni ensin tarkastaa onko sen nykyinen kohde vielä käypä. Kohde on käypä, jos a) tornilla on olemassa oleva kohde, b) se on sekä tornin näköetäisyydellä *radius* (tornin ja vihollisen välinen etäisyys on pienempi tai yhtäsuuri kuin *radius*) ja c) kohde on vielä hengissä. Jos nykyinen kohde on käypä, ei tornin tarvitse vaihtaa kohdettaan. Muussa tapauksessa, jos kohde ei ole käypä, unohtaa torni ensin nykyisen kohteensa, jos se on olemassa ja alkaa etsimään uutta. Etsiessään uutta torni käy läpi jokaisen vihollisen ja löytää ensimmäisen vihollisen, joka on tornin näköetäisyydellä (kuten yllä) ja asettaa tämän nykyiseksi kohteekseen.

(**Huom**: *Tässä kohtaa olisi mahdollista toteuttaa älykkäämpi algoritmi, jolloin tornille saataisiin kehitettyä tekoäly. Tornin olisi mahdollista jonkin ehdon mukaan ehdon parhaiten täyttävä, esimerkiksi vahvin mahdollinen vihollinen kohteekseen. Todellisuudessa viholliset syötetään torneille peliä päivitettäessä 'uutuus-järjestyksessä', eli siinä järjestyksessä, missä ne on peliin lisätty, jolloin tornit yrittävät luonnostaan tähdätä 'vanhimpia' vihollisia, mikä toimii keskivertona ratkaisuna.)*

**Projektiilien osuminen**

Joka päivityksellä jokainen projektiili vertaa itseään jokaiseen viholliseen. Projektiili havaitsee osuman, jos projektiili on tarpeeksi lähellä vihollista. Jokaiselle vihollisella on koko *size*, eli sen säde. Projektiili on tarpeeksi lähellä vihollista (osuu viholliseen) kun projektiilin ja vihollisen välinen etäisyys on pienempi tai yhtäsuuri kuin *size.* Projektiili pitää myös kirjaa vihollisista joihin se on osunut. Jos projektiili ei vielä ole osunut kyseiseen viholliseen, lasketaan se osuneeksi, vahingoitetaan vihollista ja kirjataan muistiin, että tähän viholliseen on osuttu.

**Luotiprojektiili**

Luotiprojektiilin osuessa luotiprojektiili merkitään osuneeksi ja poistetaan pelistä. Luoti-projektiilin liikkuminen on kaikista yksinkertaisin. Luotaessa luotiprojektiilia annetaan sille tieto sen omasta alkupaikkavektorista **o** ja sen kohteen paikkavektorista **t**. Luotiprojektiili pystyy laskemaan itselleen suuntavektorin **v = t - o**, joka lyhennetään sopivan kokoiseksi. Nyt luotiprojektiili voi joka päivityksellä laseka seuraavan sijaintinsa **x'** omasta sijainnistaan **x** summana **x'** = **x + v**.

**Bumerangiprojektiili**

Bumerangiprojektiilia ei poisteta sen osuessa viholliseen, vaan se poistetaan vasta kun se on palannut alkuperäiseen sijaintiinsa. Tämä on tosi, kun bumerangin nopeudelle **v** ja sen alkunopeudelle **u** pätee **v = - u**. Bumerangiprojektiilin saavuttaessa kääntöpisteensä bumerangi myös tyhjentää listan vihollisista, joihin se on osunut, jolloin palatessaan bumerangi voi osua samaan viholliseen toisenkin kerran. Bumerangiprojektiili luodaan samalla tavalla kuin luotiprojektiili ja sille lasketaan suuntavektori **v** samalla tavalla. Bumerangi myös luodin tavoin liikkuu joka kerta suuntavektorinsa verran **x' = x + v**, mutta **v** ei ole vakio. Joka päivityksellä, kun tiedetään vakiokiihtyvyys **a** lasketaan seuraava suuntavektori **v' = v - a**. Kiihtyvyysvektori **a** on alkuperäiselle suuntavektorille **u** päinvastainen ja paljon lyhempi, jolloin jokainen uusi suuntavektori **v'** on ensin edellistä lyhempi, kunnes **v'**:n pituus saavuttaa nollan eli kääntöpisteen. Tämän jälkeen jokainen **v**' on edellistä pitempi ja vastakkaissuuntainen alkuperäiselle suuntavektorille **u**.

(**Huom***: Bumerangin rata on tehty suoraviivaiseksi yksinkertaisuuden ja suoraviivaisen radan toimivuuden vuoksi. Olisi ollut myös mahdollista toteuttaa ympyrän mallinen tai muu kaareva rata bumerangille, mutta on mahdollista, että itse pelissä nämä radat saattaisivat olla huonompia.*)

**Ohjusprojektiili**

Ohjusprojektiili poistetaan pelistä sen osuessa viholliseen, mutta sen osuessa vahingoitetaan myös kaikkia lähellä olevia vihollisia. Tämä toteutuu käymällä viholliset osuttaessa uudestaan läpi ja vahingoittamalla kaikkia vihollisia, joiden etäisyys osumapisteestä on pienempi kuin ohjuksen räjähdyksen säde. Toisin kuin bumerangilla ja luodilla, ohjuksella on koko ajan tiedossa sen kohteen sijainti, paikkavektori **t**. Ohjuksella on myös nopeus v, joka on alussa 0, huippunopeus u ja kiihtyvyys a. Joka päivityksellä vihollisen uudeksi nopeudeksi lasketaan v' = min(v + a, u), eli nopeus kiihtyy vakiokiihtyvyydellä kunnes se saavuttaa huippunopeutensa u. Jokaisella päivityksellä vihollinen laskee omasta sijainnistaan **x** ja kohteensa sijainnista **t** uuden suuntavektorin **v** = **t - x** ja skaalaa tämän nykyisen nopeutensa v' suuruiseksi ja liikkuu laskemalla uuden paikkavektorinsa **x' = x + v**. Näin ohjusprojektiili hakeutuu kohdettaan kohti. Jos käy niin, että kohde poistuu pelistä ennen, kuin ohjusprojektiili osuu kohteeseensa, jatkaa ohjusprojektiili matkaansa viimeisimmän laskemansa suuntavektorin suuntaan, mutta kuitenkin edelleen kiihdyttäen.

**Pelin päivittäminen**

Viimeinen mainitsemisen arvoinen algoritmi on pelin päivittäminen, jonka pseudokoodi löytyy seuraavalta sivulta. Ideana pelin logiikassa on, että jokainen pelin vihollinen, torni ja projektiili käydään erikseen läpi ja niille tehdään tarpeelliset päivitykset, käyttäen avuksi tässä edellä mainittuja algoritmeja. Tämän jälkeen yritetään kutsua tältä aallolta uusia vihollisia sisään ja lopuksi palkitaan pelaaja jos nykyinen aalto on voitettu.

*Huomioita pseudokoodista:*

Todellisuudessa funktiota ja useita sen kutsumia funktioita kutsutaan muuttujalla elapsedTime, joka kertoo, kuinka paljon aikaa on edennyt ja kuinka suureksi liikkeet tulisi skaalata.

Riveillä 23-24 poistetaan torneja jotka on päivitetty. Tämä johtuu siitä, että kun torni päivitetään, luodaan samaan kohdaan uusi torniolio ja asetetaan vanhan tornin *upgraded* muuttuja todeksi, jolloin vanha torni voidaan poistaa alta.

Käymällä silmukka läpi useaan kertaan nopeutettaessa saadaan peliä nopeutettua ilman, että nopeutus aiheuttaa muutoksia pelin luonteessa.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

player = Player-object

wave = Wave-object

enemies = Array of enemy-objects

towers = Array of tower-objects

projectiles = Array of projectile-objects

function updateGame = {

loop more than once if fastforwarding, else once {

for each enemy in enemies: // Updating all enemies

if enemy is dead:

reward player (by enemy's reward amount)

remove enemy from enemies

for each spawn of enemy:

add spawn to enemies

else:

move enemy

if enemy has reached end:

damage player (by enemy's damage amount)

remove enemy from enemies

for each tower in towers : // Updating all towers

if tower was upgraded:

remove tower from towers // New tower already exists

update tower's target (with enemies)

if tower shoots:

add tower's projectiles to projectiles

for each projectile in projectiles: // Updating projectiles

move projectile

try to collide projectile with enemies

if projectile is finished :

remove projectile from projectiles

if wave is not finished spawning: // Spawning enemies to the game

if wave spawns on this update:

add spawn to enemies

if wave is cleared (all enemies dead):

reward player (by wave's reward amount)

# Tietorakenteet

Suurin osa pelin tiedoista löytyy *Game*-olion sisältä, josta löytyy buffer-tyyppiset kokoelmat vihollisille, torneille, projektiileille ja pelin koristeille. Vihollisille, torneille ja projektiileille buffer oli hyvin luontainen valinta, sillä olioita poistetaan ja lisätään peliin alituisesti ja muuttuva kokoelmatyyppi sopii tähän tarkoitukseen hyvin.

Pelioliossa on myös viittaus aina yhteen Wave-tyyppiseen aalto-olioon, jotka ladataan yksi kerrallaan tiedostosta aina aallon vaihtuessa. Aallon sisällä on sen aallon viholliset tallennettu jonoon. Jono on hyvin luontainen tietorakenne tähän, sillä aalloissa viholliset ovat ikään kuin jonossa, jossa aallossa sijaitsevan kellon mukaan jonon ensimmäinen päästetään pois jonosta ja lisätään peliin. Jonojen *dequeue()* metodi sopii siis loistavasti tähän.

Pelioliossa on myös viittaus aina yhteen polkuun, joka on poluista ensimmäinen. Polut ovat itse tehty yksisuuntainen, ketjumainen tietorakenne. Jokaisessa polussa on viittaus siitä seuraavaan polkuolioon. Tämä tekee vihollisten liikkumisen hyvin helpoksi ja määrää selkeästi poluille yksiselitteisen suunnan ja järjestyksen. Poluilla on toki myös *toArray()* metodi, joka rakentaa polusta itsestään ja kaikista sitä ketjussa seuraavista poluista taulukon tiettyihin erikoiskäyttöihin.

Jokainen vihollisolio sisältää iteraattorin, joka sisältää kaikki vihollisen kuollessa syntyvät viholliset. Iteraattorin valinta oli tässä osittain mielivaltainen, sillä taas kokoelman tyypillä ei juurikaan ollut väliä. Jälkeenpäin ajatellen, aaltojen tavoin jono olisi saattanut olla toinen luonteva kokoelmatyyppi.

Käyttöliittymän puolella ei ole mainittavia tietorakenteita. Käyttöliittymän kaikki tieto-rakenteet ovat ScalaFX-kirjaston määräämiä ja liittyvät vain käyttöliittymän toiminnalli-suuksiin ja yksityiskohtiin.

# Tiedostot ja verkossa oleva tieto

Kaikki ohjelman käyttämät tiedostot ovat XML-tiedostoja, jotka ovat projektissa kansiossa src/data valmiina, eikä niihin tarvitse (tai välttämättä edes kannata) erikseen koskea. Peli ei käsittele missään vaiheessa verkossa olevaa tietoa.

**Tiedostot**

Ensimmäinen tiedosto, defaultdata.xml sisältää yksittäisen, tyhjän pelin kentän. Tästä tiedostosta luetaan pelin ainoa valmiiksi rakennettu vakiokartta, kun päävalikosta halutaan aloittaa uusi peli. Tämä on erillään muista tasoista, ettei sitä vahingossakaan pyyhittäisi. Muut tasot ovat myös luonteeltaan erilaisia, sillä niitä poistetaan ja lisätään käyttäjän pyynnöstä ja ne ovat täysin käyttäjän itsensä luomia.

Toinen tiedosto, savedata.xml sisältää yksittäisen pelin, jossa on sekä kenttä että pelin sen hetkinen tilanne, sisältäen tornit ja nykyisen aallon numeron. Tähän tiedostoon tallennetaan aina viimeiseksi tallennettu peli ja tästä tiedostosta ladataan, jos halutaan jatkaa viimeksi tallennettua peliä.

Kolmas tiedosto, customdata.xml sisältää mahdollisesti useita tyhjiä kenttiä. Tänne tallennetaan kaikki käyttäjän itsensä level editorissa luomat kentät.

Neljäs ja viimeinen tiedosto on waves.xml, mihin on ennalta määritetty jokaisella pelin aallolla syntyvät viholliset. Tähän tiedostoon lisäämällä aaltoja peliin voi lisätä enemmän aaltoja pelattaviksi.

**Tiedostojen rakenne**

Tiedostot defaultdata.xml, savedata.xml ja customdata.xml käyttävät kaikki samaa formaattia pelien tallentamiseen. Pelit on kirjattu <game> elementteihin. Tiedosto customdata.xml on ainoa, joka sisältää useita eri <game> elementtejä, joten siinä missä defaultdata.xml:llä ja savedata.xml:llä <game> on juurielementti, on customdata.xml:llä vielä <data> elementti juurielementtinä, jonka sisään on suljettu useita <game> elementtejä.

Pelien ominaisuudet on <game> elementtien sisällä omissa elementeissään. Pelikentän koko (joka päätyikin kehityksessä olemaan muuttumaton) on tallennettu <cols> ja <rows> elementteihin yksittäisinä numeroina. Nykyisen aallon numero on tallennettu <wave> elementtiin yksittäisenä numerona. Pelaajan tiedot on <player> elementtiin kirjoitettuna <health> ja <money> elementeissä numeroina. Tornit ovat <towers> elementissä, jossa jokainen torni on <tower id, x, y/> elementtinä ja sisältää oman tunnuksensa id, jonka avulla voidaan tunnistaa, minkä tyyppinen torni on kyseessä ja sijaintinsa x ja y muuttujissa.

Polku on <path> elementissä yksittäisinä <segment x, y/> elementteinä, joista jokaisella on oma sijaintinsa x, y kokonaislukuina. Polkusegmenttien järjestyksellä on väliä, sillä ensimmäinen polkusegmentti on polun alkupiste ja viimeinen loppupiste ja polun segmenttien järjestys on sama kuin tiedostossakin. Viimeisenä koristeet löytyvät <props> elementistä <prop id, x, y/> elementtenä samankaltaisesti kuin tornitkin. Tyhjä peli (kuten tiedostoissa customdata.xml ja defaultdata.xml) ei sisällä ollenkaan torneja <towers> elementissä ja sen aallon numero on 0. Vain savedata.xml sisältää keskeneräisiä pelejä.

Aallot on tallennettu waves.xml tiedostoon, jossa juurielementti <waves> sisältää useita <wave, prize> -elementtejä siinä järjestyksessä, joissa ne pelissä tulevat vastaan. Jokainen aalto sisältää rahasumman prize, jonka pelaaja saa voitettuaan aallon ja <enemies> elementin, joka sisältää useita <enemy count, id/> elementtejä, jossa id kertoo aaltoon kuuluvien vihollisten tyypin ja count lukumäärän.

# Testaus

Ohjelman suuri puute: ohjelmaa ei ole yksikkötestattu. Tämä johtuu suurelta osalta siitä, että yksikkötestien laatiminen on haastavaa tämän tyyppiselle pelille. Jokaisen luokan perusominaisuudet olivat toiminnallisuuksiltaan niin yksinkertaisia, etteivät ne vaadi yksikkötestauksia. Luokkien monimutkaisemmat ominaisuudet (pääasiassa kohdassa 4 luokitellut algoritmit) olisivat saattaneet vaatia yksikkötestausta. Yksikkötestaukset vaativat kuitenkin, että rakennetaan uusi peli ja peliin luodaan vihollisia, torneja ja/tai projektiileja. Suurin osa testattavista ominaisuuksista koskee juuri näiden luokkien välisiä interaktioita. Peliä olisi pitänyt päivittää ja osata ennustaa, miten asioiden kuuluu lopuksi olla. Tätä vaikeuttaa vielä se, että testattavat ominaisuudet ovat ajasta riippuvaisia. Tämänkaltaisten testien rakentaminen pelille tuntui kuitenkin sen verran haastavalta verrattuna siihen, että rakensi yksinkertaisen graafisen käyttöliittymän hyvissä ajoin ja peliä testasi sitä pelaamalla. Oikeanlaisen pelin kulun ja olioiden välisen interaktion pystyi selkeästi havaitsemaan manuaalisesti testaamalla käyttäen graafista käyttöliittymää.

Käyttöliittymän yksikkötestaus olisi vielä hankalampaa, mutta käyttöliittymä on toiminnalli-suudeltaan melko yksinkertainen ja sen manuaalinen testaaminen oli moninkertaisesti helpompaa ja nopeampaa kuin itse pelin testaaminen.

Tiedostojen välistä interaktiota ei myöskään ole yksikkötestattu, mutta interaktio on vahvistettu toimivaksi. Pelin ja tiedostojen välinen interaktio on useimmiten vain tiedostoista lukemista, jonka toiminta on helppo vahvistaa oikeaksi, ja jonka epäonnistuessa saa virheen. Kuitenkin muutamassa kohdassa, jossa tiedostoihin myös kirjoitettiin, on pystytty kirjoittaminen toteamaan toimivaksi, sillä tiedostossa on aina lukenut juuri se, mitä siellä on pitänyt lukea ja ohjelma on myöhemmin pystynyt tiedostoja virheettömästi myös lukemaan.

Tämä vastasi melko hyvin sitä mitä suunnitelmassa esitettiin.

# Ohjelman tunnetut puutteet ja viat

Eräs tunnettu, mutta onneksi hyvin epätodennäköinen vika syntyy pelisäikeen ja käyttö-liittymän rinnakkaisuudesta ja pelisäikeen itsenäisestä ajanmittaamisesta. Tämän on havaittu muutaman kerran aiheuttavan virhetilanteita joko käyttöliittymässä, etenkin peliä piirrettäessä ja/tai pelissä itsessään. Näitä virheitä on korjattu suuria määriä, mutta siitä huolimatta on mahdollisuus harvinaisissa tilanteissa saada jokin selittämätön virhe. Näiden virheiden korjaaminen on hankalaa, sillä niiden syytä on vaikea löytää ja ne ovat erittäin harvinaisia ja niiden tahallinen uudelleen aiheuttaminen on käytännössä mahdotonta.

Puutteiksi voi myös laskea pelin sisällön vähäisyyden: pelissä on vain kolmea tornia, joista jokaiselle on vain muutama päivitystaso, aaltoja on vain reilu kaksikymmentä ja vihollis-tyyppejä on vain neljä. Pelissä myös pelin eri ominaisuudet eivät ole pelin kannalta tasa-painoisia ja toiset pelityylit ovat varmasti tästä johtuen ylivoimaisia, joskin vihollisten taso on yritetty saada sopivan haastavaksi.

korjaa se vitun f11

upgradet yläreunassa

Kun paino tarpeeks kauan pohjassa nopeutusta niin pamausäänet katos :(

Itseasias kaikki äänet paitsi musiikki

# Kolme parasta kohtaa

Leipäteksti.

# Kolme heikointa kohtaa

Leipäteksti.

# Suunnitelman ja aikataulun toteutuminen

Leipäteksti.

# Kokonaisarvio lopputuloksesta

Leipäteksti.

# Viitteet

**Scala Documentation** <https://docs.scala-lang.org/>

**Scala Standard Library API** <https://www.scala-lang.org/api/current/>

**ScalaFX** <http://www.scalafx.org/>

**ScalaFX API** <http://www.scalafx.org/api/8.0/index.html>

**JavaFX API** <https://docs.oracle.com/javase/8/javafx/api/toc.htm>

**StackOverflow** <https://stackoverflow.com/>

**Mark Lewis** <http://www.cs.trinity.edu/~mlewis/>

**Scala XML** <https://github.com/scala/scala-xml>

**Alvin Alexander, Scala** <https://alvinalexander.com/scala>

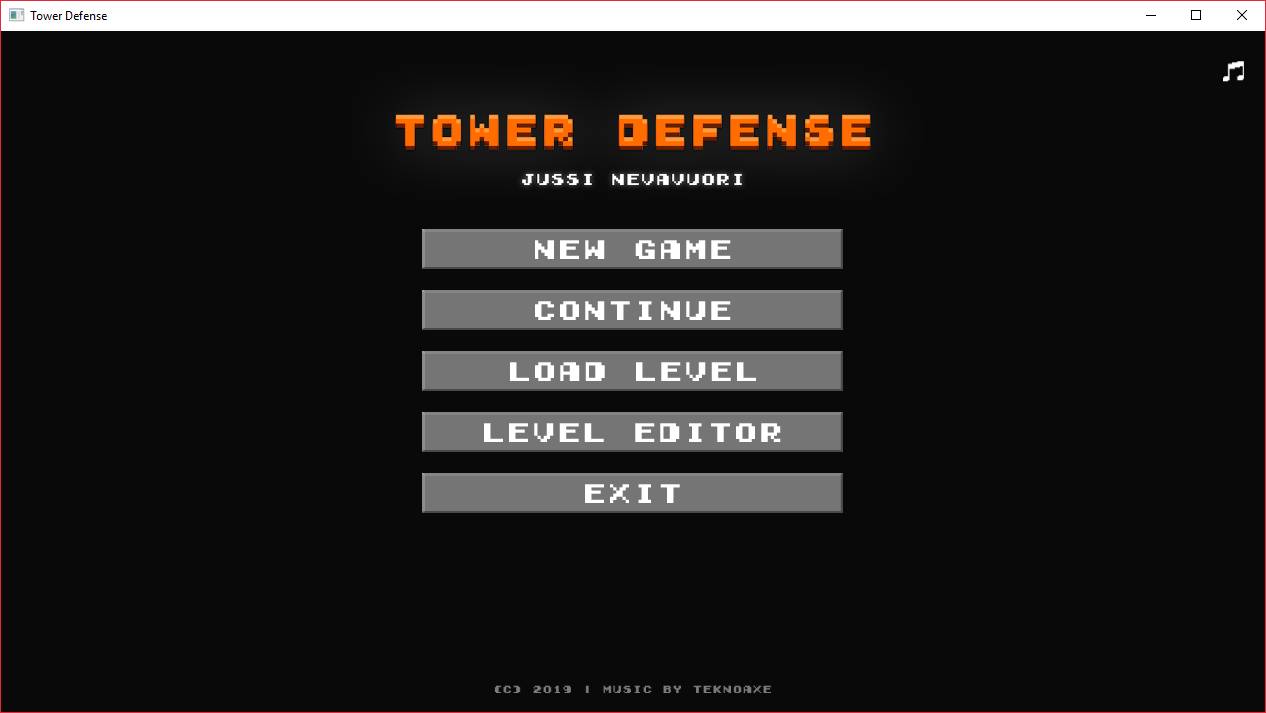
# Liitteet

Liitteenä koko projektin lähdekoodi ja ulkoiset kirjastot .zip -tiedostossa.

# Näyttökaappauksia



Peli eri vuodenaikoina

 Pelin päävalikko

Tasonvalintavalikko



Level Editor