Ohjelmistoarkkitehtuuri ja mobiilipelien kehitys

Henrik Rossi

Haaga-Helia ammattikorkeakoulu
Tradenomi
Opinnäytetyö
2023

Tiivistelmä

Tekijä(t)
Henrik Rossi

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Ohjelmistoarkkitehtuuri ja mobiilipelien kehitys

Sivu- ja liitesivumäärä
24 + 1

Tämän opinnäytetyön tavoite oli toteuttaa mobiilipeli Unity-pelimoottorilla ja C# kielellä. Pelin toteutuksessa keskityttiin erityisesti koodikannan selkeyteen, arkkitehtuuriin ja jatkokehittämismahdollisuuksiin. Tekijällä oli työtä aloittaessa entuudestaan taustaa pelikehityksestä, mutta ei ohjelmistoarkkitehtuurillisesta näkökulmasta. Peli kehitetään projektin aikana proof of concept tasolle.

Ennen teoriaosuutta käydään läpi ohjelmisto- ja pelikehitykselle keskeisiä käsitteitä.

Teoriaosuudessa käsitellään mitä ohjelmistoarkkitehtuuri on ja mikä on sen merkitys pelikehityksen kontekstissa. Tässä osuudessa myös syvennytään tarkemmin suunnittelumalleihin ohjelmistoarkkitehtuurin osa-alueena ja tarkastellaan esimerkkisuunnittelumalleja.

Toiminnallisessa osassa käydään läpi toteutetun pelin kehitysprosessi vaiheittain. Ensin kuvaillaan projektin toteutusmenetelmä ja suunnitteluvaihe. Tämän jälkeen esitellään projektin toteutus sprinttikohtaisesti. Sprintit on kuvattu päiväkirja tyyppisesti ja niissä esitellään kunkin sprintin tavoitteet, toteutus ja tulokset. Lopuksi toteutetun pelin tarkemmassa esittelyssä keskitytään erityisesti pelin lähdekoodin rakenteisiin ja arkkitehtuuriin.

Loppupohdinnassa käydään läpi, kuinka hyvin työn tavoitteissa onnistuttiin sekä tarkastellaan tekijän oppimia asioita ja kokemuksia työn toteutuksen ajalta.

Tässä työssä ei keskitytä toteutetun pelin visuaaliseen ilmeeseen tai audiomaisemaan. Työn koodikantaa ei myöskään käsitellä tarkasti, vaan sen toimintaa tarkastellaan konseptien ja mallien avulla ohjelmistoarkkitehtuurin näkökulmasta. Tämä työ ei myöskään syvenny tarkemmin C#-kieleen tai Unityn käyttöön pelimoottorina.

Asiasanat
Pelikehitys, Ohjelmistoarkkitehtuuri, Unity

Sisällys

1. Johdanto ... 1
2. Ohjelmistojen arkkitehtuurit sekä suunnittelumallit ... 3
    2.1 Ohjelmistoarkkitehtuuri ... 3
    2.2 Suunnittelumallit ... 4
    2.3 Pelien ohjelmistoarkkitehtuuri ja suunnittelumallit ... 6
    2.4 Keskeiset suunnittelumallit ... 7
        2.4.1 Tila ... 7
        2.4.2 Objektivarasto ... 7
3. Toteutettava peli ... 8
    3.1 Toteutusmenetelmä ... 9
    3.2 Sprintti 1 ... 10
    3.3 Sprintti 2 ... 11
    3.4 Sprintti 3 ... 12
    3.5 Sprintti 4 ... 13
    3.6 Sprintti 5 ... 14
4. Toteutetun pelin tarkempi kuvaus ... 16
5. Pohdinta ... 22
6. Lähteet ... 24
Liitteet ... 25
    Liite 1. Linkki projektin GitHub-repositorioon ... 25
    Liite 2. Linkki ladattavan pelin sivulle ... 25
    Liite 3. Linkki videoon pelistä toiminnassa ... 25

1. Johdanto

Toteutan tässä opinnäytetyössä yksinkertaisen mobiilipelin. Minulla on aikaisempaa kokemusta peliohjelmoinnista, mutta en ole aikaisemmin toteuttanut pelin koodikantaa siten, että se noudattaisi hyviä arkkitehtuurin periaatteita. Siksi keskityn tämän projektin toteutuksessa siihen, miten peli toteutetaan hyvin ohjelmistoarkkitehtuurin näkökulmasta. Syvennyn myös ohjelmistoarkkitehtuuriin aiheena, mitä se on ja miksi se on tärkeää.

Peli toteutetaan Unity-pelimoottorilla ja kielenä käytetään C# ohjelmointikieltä. Käytän toteutetun pelin mallina ja esikuvana jo olemassa olevaa peliä, jotta voisin säästää resursseja minimoimalla suunnitteluvaiheeseen käytetyn ajan. En toteuta peliin grafiikka- tai audioassetteja, mistä johtuen pelin ulkoasu tulee näyttämään hyvin minimalistiselta. Pelissä tulee olemaan myös verrattavan vähän pelattavaa sisältöä, mutta toteutetun arkkitehtuurin on tarkoitus mahdollistaa uuden sisällön tuottaminen vaivattomasti.

Minulla on aiempaa kokemusta pelikehityksestä koulutukseni ja harrastukseni puolesta. Olen opiskellut pelikehitystä ennen opintojani Haaga-Heliassa. Lisäksi olen harrastanut pelien kehittämistä omalla ajallani. Vaikka olen pääasiallisesti toiminut peliprojekteissa ohjelmoijana, minulta löytyy myös kokemusta muiltakin pelikehityksen osa-alueilta, kuten pelisuunnittelusta.

Tutkin tässä työssä ohjelmistoarkkitehtuuria ja erityisesti erilaisia suunnittelumalleja. Pelien ohjelmistoarkkitehtuuri noudattaa suureksi osaksi samoja periaatteita kuin muutkin ohjelmat, mutta tutkin ohjelmistoarkkitehtuuria juuri pelien näkökulmasta. Tutkin erilaisia suunnittelumalleja, jotka tukisivat pelin tavoitetta olla helposti jatkokehitettävä. Esittelen myös tarkemmin suunnittelumalleja, joita olen käyttänyt toteutetussa työssä.

Opinnäytetyössä ei keskitytä siihen, kuinka pelejä ohjelmoidaan yleisesti. Kaikki ohjelmistoarkkitehtuuriin liittyvät mallit ovat sovellettavissa myös muissa ympäristöissä ja kielissä. Työ ei siksi sisällä tarkempaa lähdekoodin tarkastelua.

Projektin onnistumista mitataan pääasiassa toteutetun arkkitehtuurin selkeyden ja jatkokehitettävyyden perusteella. Toteutetun pelin päämekaniikoiden tulee myös toimia. Pelin grafiikkaa tai audiopuolta ei oteta huomioon arvostelussa.

Tämä opinnäytetyö on merkittävä, koska se tarjoaa tietoa ohjelmistoarkkitehtuurin tärkeydestä ja mitä hyötyjä se tarjoaa peli- ja ohjelmistoprojekteissa. Tämä työ voi toimia myös inspiraationa muille pelikehittäjille, jotka haluavat viedä pelinsä koodikannan seuraavalle tasolle.

Keskeiset Käsitteet

Sprintti	Yksittäinen kehityssykli, jonka aikana suunnitellaan ja toteutetaan jokin tavoite. Sprintin päätteeksi pidetään retroperspektiivi, jossa arvioidaan sprintin onnistumista (Schwaber & Sutherland, 2020).
Skripti	Koodi, joka kontrolloi peliobjekteja ja -tapahtumia. Yleensä kirjoitettu jollain korkeamman tason ohjelmointikielellä. (Game Industry Career Guide, 2023)
Peliobjekti	Kaikki peliin kuuluvat yksittäiset asiat kuten hahmot, esineet, kamera tai efektit. Peliobjekti ei tee itsessään mitään mutta sille voidaan määrittää erilaisia ominaisuuksia ja toimintoja. (Unity User Manual, 2022)
Instanssi	Yksittäinen ilmentymä jostain objektista tai luokasta. Jonkin asian instanssin muokkaaminen ei muokkaa muita samanlaisia instansseja. (Tech Target, 2022)
Scene	Muokattava alue pelissä. Koostuu erilaisista objekteista, jotka muodostavat yhdessä pelattavan ympäristön. (Unity Game Deveplopment Terms, 2023)
Assetti	Termi, jolla kutsutaan kaikkia asioita pelissä, kuten peliobjekteja, 3D-malleja tai ääniä. (Unity Game Deveplopment Terms, 2023)
Placeholderi	Väliaikainen assetti, jota käytetään siihen asti, kunnes lopullinen versio saadaan valmiiksi (Placeholder Contents in Game Development, 2015)
Syöte	Syötteeksi (input) kutsutaan kaikkia tapoja, joilla pelaaja vuorovaikuttaa pelin kanssa. Tyypillisesti pelaajan syöte pelissä tapahtuu näppäimistön, peliohjaimen tai mobiililaitteen sensoreiden välityksellä. (GameMaker Manual, 2023)

2. Ohjelmistojen arkkitehtuurit sekä suunnittelumallit

2.1 Ohjelmistoarkkitehtuuri

Bass, Clements ja Kazman (2013, 19) mukaan jokaisella ohjelmalla on arkkitehtuuri, oli se sitten hyvin tai huonosti organisoitu. Koostui ohjelma sitten yhdestä tai tuhannesta komponentista, voidaan sillä katsoa olevan oma arkkitehtuurinsa. Kaikilla systeemeillä on jokin sisäinen logiikka, joka määrittää missä mikäkin on ja mihin mikäkin asia vaikuttaa. Nystromin (2014, 9) mukaan ohjelmistoarkkitehtuurissa onkin kyse juuri sen tarkastelusta, miten ohjelman koodi on organisoitu, eikä niinkään miten tai millä kielellä se on kirjoitettu. Bassin ja kollegansa (2013, 18) mieltävätkin ohjelmiston arkkitehtuurin olevan joukko ohjelman rakenteita.

Teknisesti hyvinkin vaativa ohjelma saattaa olla pellin alle katsoessa huonosti organisoitu, eikä tämä välttämättä vaikuta mitenkään loppukäyttäjän kokemukseen. Nystrom (2014, 18) väittääkin, että ohjelman arkkitehtuurilla ei ole merkitystä, jos sen koodia ei tarvitse koskaan muokata jälkeenpäin. Kuitenkin tällainen tilanne on harvoin todellisuutta. Nykyään ohjelmistokehitys ei koostu vain tuotteen kehityksestä ja valmiin tuotteen myymisestä sellaisenaan. Ohjelmistot ovat monimutkaisia ja elinkaarensa aikana jatkuvasti kehittyviä systeemejä. Kun ohjelmaan halutaan lisätä uusia ominaisuuksia tai korjata vanhoja, sen koodia täytyy käydä läpi ja muokata. Näissä tilanteissa sotkuisen ja huonosti organisoidun koodin läpikäyminen on hidasta ja siksi kallista.

Vaikka selkeämpi koodi on hyvin tärkeä osa ohjelmistoarkkitehtuuria, se ei ole ainoa ominaisuus mihin kiinnitetään huomiota. Ohjelma koostuu erilaisista elementeistä, jotka voivat olla toisistaan riippuvaisia ja siten sidottuja toisiinsa. Nystrom (2014, 18) kuvailee, että sidonnaisuus on sitä, kun elementtejä tutkiskellessa toista ei voida ymmärtää ymmärtämättä toista. Näennäisesti kaksi eri elementtiä toimii niin tiukasti sidottuna toistensa kanssa, että toisen muokkaaminen voi haitata toisen toimintaa tai jopa estää sen kokonaan. Kuten edellä mainittu, ohjelmistokehitys on jatkuva prosessi ja ohjelman lähdekoodia täytyy muokata jatkuvasti. Mitä enemmän tällaisia edellä kuvattuja tiukkoja sidonnaisuuksia on lähdekoodin sisällä, sitä suurempi riski jokaisella koodin muokkauksella on rikkoa jotain. Kun muokkaa kohtaa A, päätyy rikkomaan kohdan B, joka vuorostaan rikkoo C ja D ja niin edelleen. Tämä voi lisäksi tapahtua täysin huomaamatta, jos eri elementtien väliset sidonnaisuudet eivät ole tarpeeksi selkeitä ja johdonmukaisia. Tärkeä osa ohjelmistoarkkitehtuurissa onkin tällaisten sidosten vähentäminen tai purkaminen. Ohjelman eri elementeistä voidaan esimerkiksi tehdä enemmän modulaarisia, milloin yksi elementti vastaa vain yhdestä tehtävästä. Nystrom (2014, 20) haluaa kuitenkin täsmentää, että kaikkien koodin sidonnaisuuksien poistaminen ei myöskään ole ideaalia. Tällaisen sidonnaisuuksista puhtaan koodin tuottaminen ei ole aina kustannustehokasta ja sen lisäksi se ei ole myöskään välttämättä selkeää tai helppolukuista. Kun toteutettu oikein, sidonnaisuudet luovat koodille merkitystä ja auttavat ymmärtämään sen

toimintaa. Ohjelmistoarkkitehtuurissa on tärkeää löytää toimiva keskitie projektikohtaisesti sidonnaisuuksien ja niiden välttämisen välillä.

Koska ohjelmiston kehityskulut voivat kasvaa liki eksponentiaalisesti sen elinkaaren aikana, on hyvän arkkitehtuurin suunnitteleminen tärkeää jo aikaisessa vaiheessa. Ongelmalliset rakenteet koodikannassa aiheuttavat komplikaatioita ja lisätyötä jokaisella ohjelmiston kehitysiteraatiolla. Esimerkki tällaisesta voisi olla luokka X. Tämän luokan kanssa keskustelevat luokat joutuvat kääntämään lähetettyä dataa tälle luokalle sopivaksi. Tämä ei välttämättä muodostu ongelmaksi ohjelman kehityksen alkuvaiheessa, kun luokan X kanssa keskustelevia luokkia on vähän. Kun muita luokkia lisätään yhä enemmän, joudutaan jokaisen uuden luokan kohdalla tekemään lisätyötä X luokan kanssa keskustelun mahdollistamiseksi. Tässä tapauksessa tilanteen voisi esimerkiksi ratkaista lisäämällä luokkaa X metodin, jonka avulla se kääntää sille lähetetyn datan itse, eliminoimalla tämän tarpeen muilta luokilta.

Koskimies ja Mikkonen (2005, 18) vertaavat ohjelmistoarkkitehtuuria perustuslakiin. Hyvin suunniteltu ohjelma on rakenteiltaan ja käytännöiltään johdonmukainen ja looginen. Kehittäessä ohjelmaa eteenpäin aikaisemmin tehtyjen hyvien suunnitteluvalintojen ns. ”lakien” noudattaminen varmistaa, että ohjelma pysyy jatkossakin ymmärrettävänä ja helpommin jatkokehitettävissä.

Yhteenvetona, ohjelmistoarkkitehtuurin tarkoitus on varmistaa, että ohjelman lähdekoodi on mahdollisimman hyvin ymmärrettävissä. Se myös pyrkii pitämään sen sisäisten logiikan johdonmukaisena siten että oikeat asiat ovat sidottuina toisiinsa ja oikeat elementit pidetään irrallaan. Huonosti suunniteltu ohjelmistoarkkitehtuuri voi johtaa ohjelmaan, jonka lähdekoodin muokkaaminen ja jatkokehittäminen on hidasta, siksi kallista ja joskus jopa mahdotonta.

2.2 Suunnittelumallit

Suunnittelumallit ovat tämän työn kohdalla keskeisessä asemassa, sillä työn toiminnallisessa osassa tutkin ja implementoin erilaisia malleja toteutettuun peliin. Suunnittelumallit ovat myös yksi tärkeä osa ohjelmistoarkkitehtuuria.

Ohjelmistokehityksessä, kuten muillakin aloilla, on tärkeää välttää keksimästä pyörää uudestaan. Kuten hyvien ideoidenkin kohdalla on myös ongelmatilanteiden kohdalla hyvin todennäköistä, että joku toinen on kohdannut saman ongelman aikaisemmin. Bass ja hänen kollegansa (2013, 227) mieltävätkin, että on olemassa monta ratkaisua, joilla tehdä jokin asia huonosti, mutta vähän ratkaisuja, joilla tehdä se hyvin. Suunnittelumallit ovat tällaisia hyviä ratkaisuja.

Suunnittelumalli on yksinkertaisuudessaan lähestymistapa tai ohje, jolla toteuttaa jokin asia tietyllä tavalla. Koskimies ja Mikkonen (2005, 102) mukaan suunnittelumallit ovat hyväksi todettu ratkaisu tunnettuun ongelmaan. Suunnittelumallien ei ole tarkoitus ratkaista mihinkään yhteen ohjelmointikieleen liittyvää ongelmaa, vaan tarjota yleismaallisempi ratkaisu.

Suunnittelumallien kuvaus koostuu seuraavista osista: konteksti, ongelma ja ratkaisu. Käytän esimerkkinä ainokainen (singleton) nimistä suunnittelumallia kuvaamaan suunnittelumallien eri osia.

Konteksti on toistuva yleinen tilanne, jossa ongelma ilmenee. Ainokainen-mallin konteksti on, että jostain luokasta halutaan yksi ainutlaatuinen instanssi. Esimerkiksi peleissä saatetaan haluta, että pelin ääniä hallinoivaa äänimanageria on olemassa vain yksi kerrallaan.

Ongelma on yleinen ongelma, joka kuvataan sen kaikissa eri esiintymismuodoissa. Ainokainen-mallissa ongelma on, että jos luokasta luodaan useampia instansseja, se johtaa loogisiin ongelmiin. Jos käytetään aikaisempaa äänimanageria esimerkkinä, tämä tarkoittaa, että jos pelissä on useampia äänimanagereita, ne saattavat antaa äänilähteille ristiriitaisia tai päällekkäisiä viestejä.

Ratkaisu on kuvaus ongelman ratkaisevista arkkitehtuurisista rakenteista. Ainokainen-mallissa ratkaisu on, että luokalle, jota tahdotaan olevan vain yksi kerrallaan, lisätään toiminnallisuus, joka varmistaa, että kyseistä luokkaa on vain yksi kerrallaan olemassa. Käytännössä tämä tarkoittaisi sitä, että äänimanagerilla olisi ominaisuus, jolla se tuhoaa itsensä, jos se havaitsee muita luokkansa edustajia pelissä. Tämä johtaa siihen, että viimeinen jäljellä oleva instanssi äänimanagerista jää yhdeksi ja ainoaksi äänimanageriksi pelissä.

Monissa suunnittelumalleissa saattaa olla yksinkertaisuudessaan kyse vain eri koodien riippuvuussuhteiden muuttamisesta, kuten esim. tietyn koodin osan siirtämisestä toiseen. Monimutkaisissa systeemeissä voi olla käytössä useampia suunnittelumalleja samanaikaisesti (Bass.ym. 2013, 229).

Saavuttaakseen joustavamman systeemin, suunnittelumalleja implementoidessa koodiin saatetaan lisätä ylimääräisiä elementtejä kuten esimerkiksi viestittimiä tai kuuntelijoita (Nystrom 2014, 21). Tällaiset elementit saattavat heikentää ohjelman suorituskykyä, mutta eivät merkittävästi (Koskimies, Mikkonen 2005, 105) ja saavutettu parempi ja joustavampi koodikanta on sen arvoinen (Nystrom 2014, 21). Nystromin mukaan ohjelmistokehitys on iteratiivinen prosessi, jossa oikeiden ratkaisujen löytäminen jokaiseen erilaiseen tilanteeseen on tärkeää. Mikään yksittäinen suunnittelumalli ei toimi yleismaallisesti kaikkiin ongelmiin ja eri mallien käyttöä kuluu harkita tilannekohtaisesti.

Hyvän arkkitehtuurin suunnittelun apuvälineenä voidaan myös käyttää antisuunnittelumalleja. Antisuunnittelumallit ovat huonoja ratkaisuja, niin sanottuja varoittavia esimerkkejä. Koskimies ja Mikkonen (2005, 107) ehdottavat että antisuunnittelumallit voivat olla hyödyllisiä auttamaan löytämään ohjelmiston arkkitehtuurin heikkoja kohtia. Tuntemalla huonon suunnittelumallin, sen syntymistä tietoisesti välttämällä voidaan päästä lähemmäksi parempaa ohjelmistoarkkitehtuuria.

2.3 Pelien ohjelmistoarkkitehtuuri ja suunnittelumallit

Pelit ovat ohjelmia siinä missä muutkin perinteisemmät ohjelmat. Peleillä on kuitenkin muihin ohjelmiin verrattuna paljon erityisiä toiminnallisia vaatimuksia, joitten takia pelien ohjelmistoarkkitehtuuri usein eroaa muista ns. tavallisista ohjelmista.

Koska pelit ovat ohjelmistollisesti niin vaativia kokonaisuuksia, on harvoin järkevää tai kannattavaa rakentaa jokaista peliä täysin tyhjästä. Suurimmalla osalla peleistä on keskenään monia samoja tarpeita kuten esimerkiksi fysiikan simulointi tai grafiikan piirtäminen. Tähän tarkoitukseen kehitetään erikseen pelimoottoreita, joiden tarkoituksena on helpottaa pelikehitystä vähentämällä näistä tarpeista syntyviä kustannuksia. Tavallisilla ohjelmistoilla ei ole välttämättä samoja vaativia ja toistuvia tarpeita kuten peleillä, jotka vaatisivat pelimoottorin kaltaisia konstrukteja. Esimerkiksi kalenterisovelluksella ei ole tarvetta 3D-grafiikan laskemiseen, sillä se käyttää ulkoasussaan yksinkertaista graafista käyttöliittymää. Tällainen sovelluksen ei myöskään tarvitse simuloida fysiikkaa, koska sen toiminnallisuus koostuu lähinnä päivämäärien asettamisesta ja muokkaamisesta. Nämä molemmat ominaisuudet ovat nykyään peleille hyvin tyypillisiä tarpeita.

Pelimoottoreiden, kuten myös tässä työssä käytetty Unity-pelimoottori, avulla pelikehittäjät voivat keskittyä pelikehityksessä siihen mikä tekee juuri heidän pelistään uniikin. Monet yritykset saattavat tuottaa omille peleilleen omat pelimoottorinsa, ja jotkut taas saattavat keskittyä yritystoiminnassaan vain pelimoottoreiden kehittämiseen tuotteena. Pelimoottoreiden käyttäminen vaikuttaa pelin ohjelmistoarkkitehtuurin suunnitteluun antamalla valmiin pohjan ja ympäristön, jonka puitteissa arkkitehtuuri myötäilee moottorin ominaisuuksia. Käsittelen aihetta myös myöhemmin, kun kerron kuinka Unity pelimoottorina vaikutti arkkitehtuurisiin valintoihin, joihin päädyin toteutetussa pelissä.

Verrattuna tavallisempiin ohjelmiin pelit sisältävät myös omaa peleille tyypillistä logiikkaa. Muilla ohjelmilla ei ole tarvetta esimerkiksi kollisioiden tunnistuksille tai monimutkaisille käyttäjän kontrolleille. Pelilogiikalle on myös ominaista sen toimiminen reaaliajassa, jossa on hyvin tarkkaa, milloin mikäkin asia tapahtuu ruuduntarkkuudella. Tämä asettaa myös pelin arkkitehtuurille vaatimuksia kuinka eri osien väliset vaikutukset ja sidonnaisuudet rakennetaan.

2.4 Keskeiset suunnittelumallit

2.4.1 Tila

Tila (State) on yksinkertainen suunnittelumalli, joka soveltuu hyvin peleihin. Gamma Helm, Johnson, Vlissides (1994, 305) mukaan tila-malli on määritetty siten, että objekti voi muuttaa toiminnallisuuttaan sen mukaan missä tilassa se on. Esimerkiksi pelaajahahmo voi olla erilaisissa tiloissa riippuen siitä, mitä pelissä tapahtuu. Hahmo voi aloittaa toimeton-tilassa ja kun pelaaja painaa juoksunappia se siirtyy juoksu-tilaan.

Tila-mallin käytöstä on paljon etuja pelien ohjelmoinnissa. Esimerkiksi tasohyppelypeleissä pelaajahahmon hyppyjen määrää pystytään rajoittamaan tila-mallilla, esimerkiksi jos hyppiminen tehdään mahdolliseksi vain tietyissä tiloissa esim. toimeton- ja juoksu-tiloissa. Myös esimerkiksi hahmon animaatio voidaan asettaa muuttumaan eri tilojen välillä.

2.4.2 Objektivarasto

Objektivarasto (object pool) on suunnittelumalli, jossa jotain resurssia kierrätetään palauttamalla resurssi takaisin varastoon sen poistamisen sijaan (Kircher, Jain. 2002, 2).

On tavallista, että peleissä luodaan ja poistetaan mittavia määriä erilaisia objekteja. Esimerkiksi yleensä, kun Unity-pelimoottorilla tehdyssä pelissä vihollinen syntyy, se luodaan Instantiate-metodilla ja kun vihollinen kuolee, se tuhotaan Destroy-metodilla. Muistin säästämiseksi on kuitenkin kannattavampaa luomisen ja tuhoamisen sijaan käyttää objektivarastoa. Objekti varastoa käyttäessä pelin alussa luodaan lista, varasto, joka täytetään tarvittavalla määrällä objekteja. Tämä määrä tulee olemaan se, kuinka monta objektia halutaan olevan enimmillään ”olemassa” yhdellä kertaa. Varastossa olevat objektit eivät ole pelin toiminnan kannalta olemassa ja ne asetetaan epäaktiivisiksi. Kun pelin aikana halutaan luoda uusi objekti, haetaankin varastosta yksi sellainen objekti ja asetetaan se aktiiviseksi. Kun objekti halutaan tuhota, sen arvot nollataan, asetetaan takaisin epäaktiiviseksi ja palautetaan takaisin varastoon.

3. Toteutettava peli

Kuva 1. Esimerkki kuva pelistä Mighty Doom (Bethesda, 2023)

Tässä opinnäytetyössä toteutan yksikertaisen mobiilipelin käyttäen Unity pelimoottoria ja C# kieltä. Peli tulee lainaamaan vahvasti toiminnallisuuttaan olemassa olevalta peliltä Mighty Doom (Bethesda, 2023). Kyseinen peli on mobiilipeli, jossa pelaajahahmo taistelee demoneita vastaan erilaisilla aseilla. Kuva 1. havainnollistaa miltä pelin toiminta näyttää käytännössä. Pelin kohderyhmää ovat toimintapelien ystävät. Heistä erityisesti ne pelaajat, jotka pitävät yksinkertaisemmista peleistä, jotka eivät vaadi isoa opettelukynnystä pelissä pärjäämiseen. Peli on suunniteltu siten, että kuka tahansa, riippumatta aiemmasta kokemuksesta toimintapeleistä, voisi pelata sitä ongelmitta.

Toteutuksen rajoittavia tekijöitä tulevat olemaan aika ja tekijöiden määrä. Molemmat näistä tulevat vaikuttamaan merkittävästi pelissä olevan pelattavan sisällön määrään, joka voi jäädä vähäiseksi. Pelin visuaaliset elementit ja äänimaisema tullaan myös jättämään kokonaan placeholdereiden tasolle.

Projektin onnistumisen tärkeimmät mittarit ovat tuotetun pelin ohjelmistoarkkitehtuurin laatu ja jatkokehityspotentiaali. Varsinkin koodin modulaarisuus on tärkeää aspekti. Modulaarinen koodi helpottaa pelin jatkokehityksessä merkittävästi, sillä se moninkertaistaa uuden pelattavan sisällön tuottamisen mahdollisuudet. Toteutuksessa ei tulla antamaan isoa painoarvoa pelin visuaaliselle

tai auditiiviselle toteutukselle. Kuitenkin näiden ominaisuuksien lisääminen pelin jatkokehityksessä tulisi olla huomioitu koodin rakenteessa.

3.1 Toteutusmenetelmä

Pelin suunnitteluvaiheelle varataan aikaa viikko. Suunnitteluvaiheen alussa tutkin esimerkkinä toimivaa peliä Mighty Doom (Bethesda, 2023) ja listaan ylös ominaisuuksia ja pelimekaniikkoja, joita tarvitaan pelin toteuttamiseen. Keskityn juuri niihin pelimekaniikkoihin, jotka tekevät mielestäni pelistä hauskan ja helposti lähestyttävän. Tutkimustyön jälkeen teen tarkemman vaatimusmäärittelyn eri ominaisuuksille ja listaan niitä tehtäviksi. Hahmotettuani paremmin kokonaisuuden karsin pois tehtävät, jotka eivät sovi projektin aikabudjettiin tai laajuuteen. Asetan tehtävät tärkeysjärjestykseen ja sen jälkeen aloitan kehitysvaiheen. Tässä vaiheessa teen myös ensimmäisen raakileen ohjelmiston arkkitehtuurista.

Kehitysvaihe tulee koostumaan viidestä viikon mittaisesta sprintistä. Sprinttien alussa määritän mitä tehtäviä toteutan sen aikana. Tyypillinen kehityspäivä tulee koostumaan uuden ominaisuuden ohjelmoinnista, testaamisesta ja virheiden korjauksesta. Testaan pelin pelattavuutta ja pelituntumaa säännöllisesti. Testautan peliä myös lähipiirissäni, jotta saisin palautetta ja uusia näkemyksiä. Sprintin lopussa käydään läpi mitä tehtäviä on saatu toteutetuiksi. Jos jokin tehtävä on jäänyt kesken, se siirretään seuraavaan sprinttiin. Pelin kehittyessä päivitän myös pelin arkkitehtuurikaaviota.

Tärkeänä tiedonlähteenä työn toteuttamiseen toimii Robert Nystromin kirja Game Programming Patterns (2014), josta löytyvät erilaiset suunnittelumallit toimivat hyvänä mallina projektin ohjelmistoarkkitehtuurille. Käytän myös erilaisia nettipalstoja ja Unity-dokumentaatiota apuna koodin ongelmia ratkoessa.

Toiminnallisesta työstä ei synny kuluja, koska työskentelen yksin eikä projektissa ole tarvetta ulkoisille resursseille, matkakuluille tai materiaalikuluille. Unity-pelimoottorin käyttö on ilmaista, kunnes sillä toteutetun pelin tuotto saavuttaa tietyn tason.

Toteutettu peli tullaan jättämään proof of concept- tasolle ja ei siksi tule sisältämään luottamuksellista tietoa, kuten käyttäjätunnuksia tai puhelimen tietoja.

Sprinttien kuvaus

Sprintti 1

Pelaajahahmon toiminnallisuus

Sprintti 2

Vihollishahmojen toiminnallisuus

Sprintti 3

Arkkitehtuurin hiominen

Sprintti 4

Pelilooppi ja koodin viimeistely

Sprintti 5

Pelattavan sisällön tuottaminen

Kuva 2. Sprinttien sisältö

Jaoin Projektin tavoitteet viidelle sprintille kuten kuvassa 2. näkyy. Sprinttien sisältö muuttui projektin edetessä useampaan otteeseen. Kuvan sprintit sisältävät viimeiset versiot siitä mitä sprintissä toteutettiin.

3.2 Sprintti 1

Sprintin 1 tavoitteeni olivat pelin perustoimintojen rakentaminen. Pelaajahahmon tuli pystyä liikkumaan, ampumaan ja suorittamaan lähitaisteluhyökkäyksiä. Tässä sprintissä myös alustin pelin Unity-projektin ja pelin ympäristön.

Pelaajahahmon liikkumiseen ei käytetä fysiikkamoottoria, vaan pelaaja liikkuu määritellyn matkan per sekunti sille annettuun suuntaan. Pelaajahahmon liikkumisen ohjaus toteutettiin virtuaalisella ohjauskepillä. Kun pelaaja painaa ruutua pohjassa sormellaan, peli määrittää tämän virtuaalisen ohjauskepille keskipisteeksi. Liikuttamalla sormea tästä pisteestä peli määrittää suunnan ensimmäisen painalluksen ja nykyisen painalluksen välillä, josta saadaan suunta, johon pelaajahahmoa liikutetaan.

Pelaajahahmon ampuminen ei vaadi pelaajalta syötettä. Kun pelaajahahmo on tarpeeksi lähellä vihollista se tähtää ja ampuu sitä automaattisesti. Tähdätäkseen vihollisia pelaajahahmo käännetään osoittamaan sille lähintä peliobjektia, jolla on vihollistagi. Toteutin ammuttavien luotiobjektien kierrätyksen objektivarasto-mallilla. Objektivarasto-malli on toteutettu seuraavalla tavalla: Luotiobjekteista luodaan pelin alussa varasto, joka sisältää isoimman määrän luoteja mitä ruudulla halutaan olevan näkyvissä yhtä aikaa. Varastossa olevia luoteja voidaan ”luoda” ottamalla varastosta

ei käytössä oleva luotiobjekti ja asettamalle se aktiiviseksi. Kun luotiobjekti osuu johonkin, se kutsuu oman varastonsa palautus-metodia, joka palauttaa sen takaisin varastoon.

Lisäsin pelaajahahmolle elämäpisteet muuttujan ja metodin, jota kutsumalla pelaajan elämäpisteet vähenevät. Jos tämän metodin lopussa hahmon elämäpisteet ovat lopussa se tuhotaan. Lisäsin myös tässä vaiheessa ominaisuuden, että kun johonkin objektiin osuu luotiobjekti se ottaa siitä vahinkoa kutsumalla edellä kuvattua metodia.

Pelaajahahmon lähitaisteluhyökkäys aktivoidaan ruudun tuplaklikkauksella. Lähitaisteluhyökkäys katsoo mitkä viholliset ovat tarpeeksi lähellä ja lähettää niille komennon ottaa vahinkoa. Tätä etäisyyttä visualisoidaan läpinäkyvällä ympyrällä pelaajan ympärillä.

Koska pelissä olevat kentät ovat pääasiassa pohjimmiltaan suoria tunneleita, pelin kameran tarvitsee seurata pelaajaa vain yhdellä akselilla. Kamera liikkuu siten että se pitää pelaajan samassa pisteessä ruudulla vertikaalisesti mutta se ei seuraa pelaajaa horisontaalisesti.

Sprintin tavoitteet suoritettiin aikataulussa, mutta näin aikaisessa vaiheessa lähdekoodi oli vielä hyvin sotkuisessa tilassa, jota sitten järjesteltiin myöhemmissä sprinteissä.

3.3 Sprintti 2

Sprintin 2 tavoitteeni oli luoda peliin vihollisobjektien toiminnallisuus. Vihollisten tulisi pystyä seuraamaan pelaajaa ja suorittamaan erilaisia hyökkäyksiä pelaajaa kohti.

Toteutin vihollisten liikkumisen Unityn omalla valmiilla reitinlaskenta-komponentilla. Unity laskee halutulle alueelle liikuttavan alueen ja esteet. Tällä alueella hahmo pystyy liikkumaan vapaasti ja liikkuessaan se kiertää mahdolliset esteet reitillään. Hahmolle määritellään kohde, jota se seuraa, tässä tilanteessa pelaajaobjekti. Koska pelin vastustajien halutaan olevan tarpeeksi helppoja pelaajalle väistellä, vihollisten liikkumisesta tehtiin hidasta ja seuraamisesta epätarkkaa. Kirjoitin itse skriptiin metodin, joka päivittää vihollisen kohteen halutulla nopeudella, että viholliset seuraisivat pistettä missä pelaaja oli hetki sitten, eikä pistettä missä pelaaja on tällä hetkellä. Tällä saavutettiin ns. zombimainen käytös vihollisille.

Kun aloin työstämään vihollisten hyökkäyksiä sain ajatuksen siitä, että viholliset voisivat käyttää samoja ampumis- ja lähihyökkäystoimintoja kuin pelaajahahmokin. Uudelleen muokkasin pelaajan ampumisen ja lähitaisteluhyökkäyksen erillisiksi skripteiksi. Pelaajahahmon skriptin tehtäväksi jäi pitää huolta vain pelaajahahmon liikkumisesta sekä vahinkopisteistä ja vihollisen skriptin omistaan.

Ampuminen ja lähitaisteluhyökkäys skriptejä pystyy erottelun jälkeen käyttämään omina komponentteina halutussa objektissa kuten pelaaja- ja vihollishahmoissa.

Sprintin tavoitteissa onnistuttiin aikataulussa, vaikkakin vihollisille päädyttiin lisäämään myöhemmin hienosäätöä pelikokemuksen parantelemiseksi.

3.4 Sprintti 3

Sprintin 3 tavoitteeni olivat muokata koodia uudelleen arkkitehtuurisesti paremmaksi. Pelin perus elementit olivat valmiita, mutta niiden jäsentelyä tulisi muokata kohti projektin päätavoitetta helposti jatkokehitettävästä pelistä.

Helpottaakseni koodin organisointia jaoin pelaaja-, vihollinen-, ampuminen- ja lähitaisteluskriptit kolmeen osaan: statistiikka, toiminnot ja perusta. Nimeäminen esim. PlayerStats, PlayerActions ja Player. Toiminnot perivät statistiikan, ja perusta perii toiminnot. Lisäksi pelaaja ja vihollinen saivat uuden kontrolleri luokan. Pelaajan syötteen käsittelylle luotiin oma manageri luokka.

Statistiikka sisältää kaikki objektin universaalisti käytetyt arvot kuten esim. kävelynopeus tai ampumishyökkäyksen vahinkomäärä.

Toiminnot sisältävät kaikki objektin päämetodit ja alimetodit kuten esim. käveleminen tai ampuminen. Päämetodit ovat mitä perusta luokka kutsuu pelin aikana suorittaakseen toimintoja, ja päämetodit kutsuvat alimetodeja toiminnon eri osien suorittamiseksi.

Perusta sisältää objektin eri toimintojen hallinnoinnin. Tämän osan toteuttamiseen hyödynnettiin tila-suunnittelumallia. Objektit, kuten esim. pelaaja, voivat olla eri tiloissa kuten toimeton, liikkuu ja kuollut. Riippuen tilastaan objekti kutsuu eri metodeja toiminnoistaan toteuttaakseen tilaansa kuuluvia toimintoja esim. liikkuessaan liikkuminen toimintoa. Perusta myös vastaanottaa kontrolleri luokalta syötettä minkä mukaan toteuttaa toimintojaan.

Käytännössä statistiikkaa ja toimintoja käytetään vain perustan kautta, jota käytetään komponenttina objekteissa. Periyttäminen näin jäsentää koodia ja helpottaa sen lukemista. Pelissä tulee olemaan myös mekaniikka, jolla pelaajan ominaisuuksia pystytään päivittämään kesken pelin, esim. pelaajan ampumistahtia voidaan nopeuttaa. Statistiikan eriyttäminen erilliseksi skriptiksi mahdollistaa sen käyttämisen statistiikkakomponenttina pelaajapäivitysobjektissa.

Kaikki pelaajan syötteeseen liittyvä koodi eriteltiin omaan syötemanageri luokkaan. Myös pelaajan syötteen käsittelyä muokattiin. Uusi versio ottaa ylös pelaajan syötteitä syöterivi muuttujiksi ja

lähettää ne oikealle komponentille syöterivilistaan. Jokainen syötteitä vastaanottava komponentti käsittelee annetun syötteen, kun se on käsitellyt listansa edellisen syötteen.

Kontrolleri luokka ohjaa kaikkia pelaaja/vihollinen, ampuminen ja lähitaistelu komponentteja, joita sen objektiin on liitetty. Pelaajan kontrolleri luokka keskustelee pelaajasyötemanageri luokan kanssa, joka ohjaa sen toimintaa. Vihollisen kontrollerilla on valmiita ohjeita, kuinka se toimii riippuen pelaajan sijainnista ja toiminnasta.

Sprintin tavoitteissa onnistuttiin ja pelin arkkitehtuuri alkoi muotoutua selkeämmäksi kokonaisuudeksi.

3.5 Sprintti 4

Sprintin 4 tavoitteena oli yhdistää aikaisemmin toteutetut perusmekaniikat pelattavaksi silmukaksi. Lisäsin myös tässä vaiheessa toiminnallisuuden, jolla pelaajan on mahdollista valita päivityksiä tasojen välissä.

Koska halusin projektin pysyvän yksikertaisena, varsinainen peli tapahtuu kokonaan yhdessä Unity scenessä. Tämä tarkoittaa käytännössä sitä, että pelin erilaiset menut ja pelattavat tasot sijaitsevat keskenään samassa tilassa.

Tein pelille yksinkertaisen menusysteemin Unityn omista nappikomponenteista. Rajoitetun aikataulun takia en lähtenyt toteuttamaan mitään hienompaa tai omaperäisempää menua, vaan menin oletusasetuksilla.

Kehitin pelin menuja ja tasoja varten ohjaamaan pelimanageri luokan. Tämä luokka hallitsee pelin kaikkia tapahtumia korkeimmalla tasolla. Myös tämän luokan toiminta pohjautuu tila-suunnittelumalliin. Pelaajan syötteet menuissa ja pelin aikana muokkaavat pelimanagerin tilaa oikeaan pelitilanteeseen. Pelimanagerilla on alaisuudessaan monia tasomanagereita, joiden toimintaa se ohjaa. Kun pelaaja aloittaa pelin tai läpäisee tason onnistuneesti, pelimanageri valitsee satunnaisen tason, jonka se aktivoi seuraavaksi tasoksi. Jos pelaaja kuolee ja häviää pelin, pelimanageri aktivoi häviö ruudun ja antaa pelaajalle mahdollisuuden aloittaa pelin alusta.

Kun pelaaja pääsee tason läpi, peli antaa pelaajalle kolme satunnaista vaihtoehtoa, joilla päivittää pelaajahahmoaan. Päivitykset voivat olla esim. ampumishyökkäystahdin nopeutus tai enemmän kestävyyspisteitä. Päivitykset ovat objekteja, joissa on komponenttina pelaaja-, ampuminen- tai lähitaistelustatistiikka. Näihin komponentteihin on helppo kirjoittaa erilaisia vaihtoehtoja erilaisista päivityksistä muokkaamalla komponentin muuttujia.

Kehitin pelin yksittäisten tasojen hallintaan tasomanagerin. Tasomanagerin tehtävä on pitää kirjaa pelaajan etenemisestä tasossa ja hallinnoida tason objekteja. Tason alkaessa tasomanageri ohjaa kaikkia sen sisäisiä vihollismanagereita täyttämään tason vihollisilla. Pelaajan etenee tasossa päihittämällä vihollisia ja tasomanageri pitää kirjaa siitä, milloin pelaaja on päihittänyt kaikki. Kun pelaaja on päihittänyt kaikki viholliset se ilmoittaa pelimanagerille, että taso on voitettu. Jos pelaaja hahmo kuolee tason aikana tasomanageri ilmoittaa pelaajan hävinneen.

Kehitin tasojen vihollisien hallintaa varten vihollismanagerin. Vihollismanagerin tehtävä on asettaa viholliset haluttuihin kohtiin kentässä tason alkaessa. Vihollismanageri on toteutettu objektivarasto-suunnittelumallilla. Pelin alussa jokainen vihollismanageri luo itselleen varaston sille määriteltyjä vihollisia ja aktivoi ja sammuttaa niitä tason aikana. Kun vihollinen kuolee, se palautetaan varastoon ja sen ominaisuudet nollataan. Vihollismanagerilla on lista koordinaateista, joihin se siirtää aktivoidessa vihollisia. Näitä pisteitä siirtämällä tasoon vihollispatterni on helppo suunnitella.

Sprintin tavoitteissa onnistuttiin, mutta vihollismanagerin toiminnassa oli vielä jäljellä ongelmia korjattavaksi.

3.6 Sprintti 5

Sprintissä 5 tavoitteena ei ollut luoda uusia ominaisuuksia peliin vaan korjata entisiä ja ennen kaikkeaan tuottaa peliin pelattavaa sisältöä.

Korjasin bugit, joita ilmeni aikaisemmissa sprinteissä vihollismanageriin ja vihollisten kuolemiseen liittyen. Monet näistä ongelmista liittyivät vihollisten ominaisuuksien nollaamiseen, jota ei tehty oikein. Varsinaiset ongelmakohdat olivat yksinkertaisia korjata, mutta ongelman lähteen löytäminen vei aikaa.

Tein peliin neljä erilaista vihollistyyppiä. Kaksi hidasta vihollista, joista toinen on heikko ja toinen kestävä. Nämä viholliset liikkuvat pelaajahahmoa kohti ja tarpeeksi lähellä ollessaan tekevät lähitaisteluhyökkäyksen. Kaksi muuta vihollista ovat ampuvia vihollisia. Toinen liikkuu nopeasti pelaaja kohden ja tarpeeksi lähellä ollessaan pysähtyy hetkeksi ja ampuu pelaajaa kohden monta ammusta. Viimeinen vihollinen liikkuu pelaajaa kohden mutta pysähtyy kauas pelaajahahmosta ampumaan niin kauan kun sillä on näköyhteys pelaajahahmoon.

Pelin kentät koostuvat seinistä ja kuiluista. Pelaajahahmo ja viholliset pystyvät ottamaan suojaa seinistä ammuntahyökkäyksiä vastaan. Seinät myös estävät kaikkien hahmojen liikkumisen niistä läpi. Kuilut toimivat muuten kuin seinät mutta eivät estä ammuntahyökkäyksiä.

Näitä elementtejä käyttämällä tein peliin neljä erilaista pelattavaa tasoa. Tein myös paljon erilaisia päivityksiä, joita pelaaja voi saada voitettujen tasojen välissä.

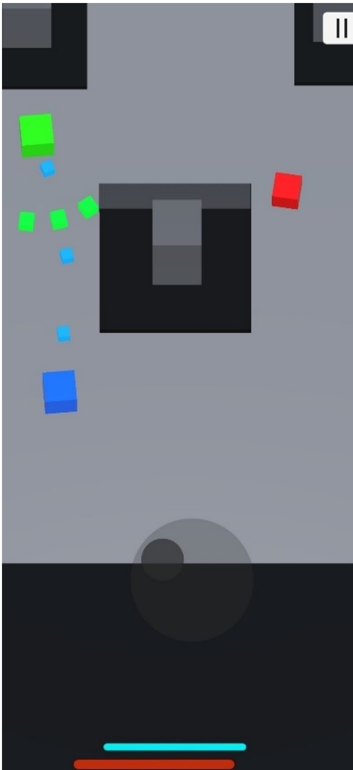
Sprintin tavoitteissa onnistuttiin ja peliin saatiin pelattavaa sisältöä. Olisin kuitenkin toivonut, että olisin ehtinyt tuottamaan enemmän pelattavaa sisältöä ja se olisi laadullisesti kiinnostavampaa. Huomasin tämän sprintin aikana, että hyvä pelidesign vaatii aikaa, kokemusta ja paljon testailua. Siitä huolimatta pelin sisältö täyttää tarkoituksensa siinä, että toteutettuja ominaisuuksia on mahdollista demonstroida käytännössä.

4. Toteutetun pelin tarkempi kuvaus

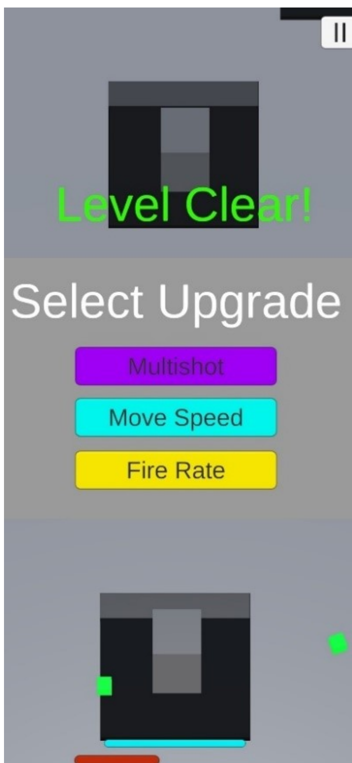
Toteutetussa pelissä pelaaja kontrolloi soturia, jolla on aseinaan ampuma- ja lähihyökkäysaseet. Pelaajan on tarkoitus eliminoida kaikki huoneessa olevat viholliset ja samanaikaisesti väistellä vihollisten luoteja ja hyökkäyksiä. Eliminoituaan kaikki huoneessa olevat viholliset pelaaja siirtyy seuraavaan samankaltaiseen huoneeseen. Tämä silmukka jatkuu, kunnes pelaaja kuolee vihollisten toimesta tai huoneet loppuvat. Pelissä on paljon erilaisia vihollisia, jotka eroavat käytökseltään ja aseiltaan toisistaan. Yksinkertaisimmat viholliset vain seuraavat pelaajaa, mutta jotkut osaavat pysähtyä kauemmaksi pelaajasta ja ampua tätä turvalliselta etäisyydeltä. Pelattavat kentät ovat yksinkertaisia huoneita, joissa on seiniä tai kuiluja, jotka vaikuttavat siihen, kuinka pelaaja ja vastustajat lähestyvät strategisesti tavoitteitaan. Kun pelaaja pääsee tason läpi hän saa palkinnoksi valita kolmesta eri päivityksestä, joka parantaa eri ominaisuuksia pelaajahahmossa esim. liikkumisnopeutta tai hyökkäysvoimaa.

Kuten pelin esikuvassa Mighty Doomissa, myös toteutetussa pelissä on käytössä yksinkertainen kontrollisysteemi, jonka käyttämiseen tarvitaan vain yhtä sormea. Tämä mahdollistaa pelin pelaamisen älypuhelimella siten, että puhelinta tarvitsee pitää vain yhdessä kädessä, menettämättä tarkempaa hallintaa pelistä. Pelaaja kontrolloi pelissä hahmonsa liikkeitä, ohjaten mihin suuntaan hahmo liikkuu. Pelaajan ohjaama hahmo ampuu lähintä vihollista itsestään ja ampumiseen tai tähtäämiseen ei tarvita ohjaamista. Tämän takia pelaajan tehtävä pelissä onkin tähtäämisen sijaan reagoida vastustajien liikkeisiin ja hyökkäyksiin väistelemällä vihollisten hyökkäyksiä. Toteutettu peli, kuten myös esikuvansa Mighty Doom, muistuttaakin tyypillisen ”Shooter”-pelin sijaan enemmän ”Bullet Hell”-peliä.

Peli on toteutettu proof of concept tasolle. Kaikki pelin pää toiminnallisuus on implementoitu, mutta assetit kuten 3D-mallit ovat vielä placeholdereita eriväristen kuutioiden muodossa. Pelin koodi on rakennettu arkkitehtuuriltaan siten, että se tukee mahdollisia tulevia animoituja 3D-malleja ja äänefektejä.



Kuva 3. Kuva pelin toiminnasta

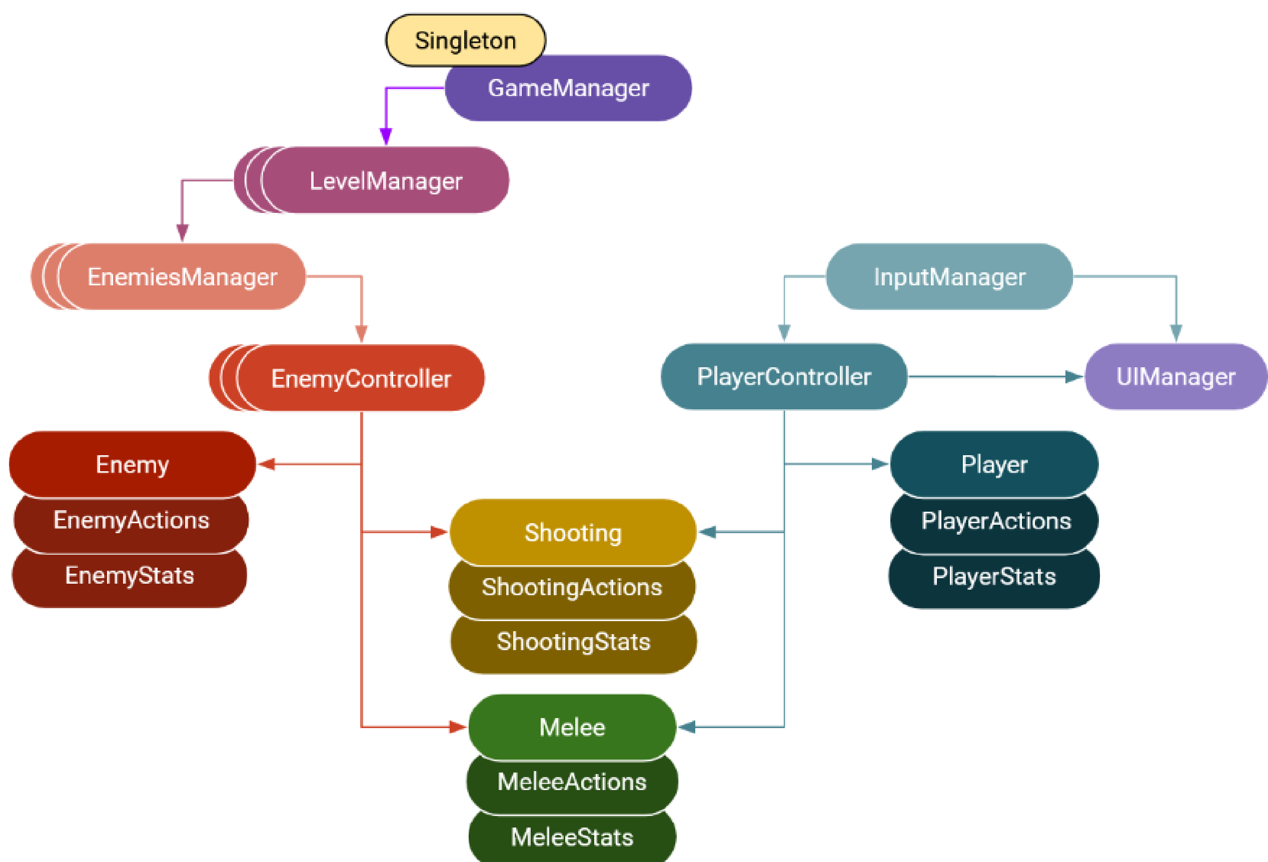


Kuva 4. Kuva tason läpäisemisestä

Kuvissa 3. ja 4. näkyy otteita pelin toiminnasta. Pelaajahahmo on sininen kuutio, joka liikkuu ja taistelee punaisia ja vihreitä kuutioita vastaan. Pelaajan käyttöliittymä koostuu punaisesta elämäpistemittarista, vaaleansinisestä energiamittarista ja harmaat ympyrät visualisoivat pelaajalle virtuaalista ohjauskeppiä. Kuvassa 4. näytetään tilanne, jossa läpäistyään tason pelaaja saa valita kolmen erilaisen päivityksen välillä.

Toteutetun pelin arkkitehtuuri on rakennettu siten että mahdollisen tulevan pelisuunnittelijan olisi helppoa toteuttaa peliin lisää sisältöä. Unity pelimoottorin editori mahdollistaa peli tasojen luomisen myös ohjelmointiin perehtymättömälle henkilölle. Pelin kaikki elementit on koostettu modulaarisista paloista, jotta kuka tahansa voisi jatkaa peliä luomalla sisältöä rakentamalla lisää vihollisia ja pelattavia tasoja näistä rakennuspalikoista. Monet arkkitehtuurivalinnat on tehty koodin puolella juuri miettien, miten koodin palasten käytöstä editorissa saisi mahdollisimman selkeää ja helppoa.

Projektin arkkitehtuurin kuvaus



Kuva 4. Toteutetun pelin arkkitehtuurikaavio

Kuvassa 4. on kuvattuna toteutetun pelin arkkitehtuurikaavio. Kaaviossa on käytetty värejä, värikirkkautta, päällekkäisyyksiä ja viivoja havainnollistamaan eri suhteita arkkitehtuurissa.

Tämä osa pelin arkkitehtuurista ei noudata mitään standardoitua mallia tai käytä valmista terminologiaa. Peliä kehittäessäni muokkasin eri luokkien välisiä suhteita vähitellen selkeämmäksi kokonaisuudeksi. Manager-luokat ohjaavat useampaa luokkaa, jotka voivat olla sijaita pelissä eri objekteissa. Controller-luokat ohjaavat vain samaan objektiin liitettyjä luokkia.

Seuraavien kappaleiden aikana esittelen tarkemmin eri osioita koodista ja niiden arkkitehtuurisia rakenteita. Useat näistä osioista on esitelty jo pintapuoleisemmin kehityssprinttien kuvauksissa.

Managerit

GameManager-luokka ohjaa pelin toimintaa kaikista korkeimmalla tasolla. Peli sijoittuu yhteen Unity sceneen, jonka tapahtumia GameManager ohjaa. GameManagerilla on eri tiloja, jotka määrittävät mitä pelissä tapahtuu milloinkin. GameManager valitsee ja aktivoi sille listatuista Level-Managereista tilanteeseen sopivan ja kommunikoi sen kanssa tason suorittamisen aikana. Pelissä on kerrallaan olemassa vain yksi GameManager-luokka ja se on siksi toteutettu ainokainen (singleton) suunnittelumallilla, jotta muilla luokilla olisi helppo yhteys siihen.

LevelManager-luokka ohjaa yhteen tasoon liittyviä asioita kuten tason tilaa ja siihen liittyviä EnemiesManagereita. Kun GameManager käynnistää tason, se asettaa ks. tason LevelManagerin tilan alkuun, joka käynnistää tason toiminnot. LevelManager asettaa pelaajan valmiiksi tasoon ja käynnistää siihen liitetyt EnemiesManagereita. EnemiesManagerit sitten spawnaavat viholliset kenttään ja pitävät LevelManagereilla kirjaa siitä, kuinka moni niistä on vielä voittamatta. Kun tason viholliset on voitettu LevelManager siirtyy tilaan voitto ja silloin GameManager tietää siirtyä eteenpäin seuraavaan vaiheeseen.

EnemiesManager luo ja ohjaa tasoon liittyviä Enemy-objekteja. EnemiesManager on toteutettu objektivarasto-mallilla kuten kuvattu edellä.

InputManager on luokka, jonka tehtävä on pitää kirjaa pelaajan antamasta syötteestä ja kommunikoida se eteenpäin PlayerControllerille ja muille luokille. Kun pelaaja antaa syötteen, InputManager koostaa sen InputLineksi ja lähettää sen eteenpäin. Riippuen syötteen tyypistä, InputLine voi sisältää infoa ruudun klikkauksesta tai virtuaalisen ohjaussauvan kääntösuunnasta. Vastaanottavalla luokalla on lista InputLinejä, joita se toteuttaa sitä tahtia, kun kerkeää.

UIManager on luokka, joka ottaa vastaan PlayerControllerilta ja InputManagerilta syötteitä, joiden perusteella muokata käyttöliittymää vastaamaan sen hetkistä pelitilannetta. UIManager esimerkiksi liikuttaa virtuaalista ohjaussauvaa pelaajan kosketuskohtaan ruudulla ja päivittää pelaajan elämäpiste mittaria.

Controllerit

Controllereiden tehtävä on keskittää yksittäisen peliobjektin modulaaristen palasien toimintaa yhdeksi entiteetiksi. Controller luokkaan liitetään kaikki halutut ominaisuudet kuten Shooting- ja Melee-luokat. Controller-luokka ohjaa näiden ominaisuuksien toimintaa muuttamalla niiden tiloja. PlayerController esimerkiksi käskee Player-komponenttia liikkumaan, kun se vastaanottaa InputManagerilta syötettä, jonka mukaan liikkua.

Player, Enemy, Shooting ja Melee

Pelin hahmojen toiminnallisuus rakennetaan näistä modulaarisista osista. Näiden koodien eri aspektit on hajautettu periytymisellä. Tämän tarkoituksena on parantaa koodin luettavuutta ja tehdä eri osioista helpommin saavutettavia. Jokaisesta moduulista on tarkoitus lisätä komponentiksi objektiin vain sen ylin osa esimerkiksi Player-luokan kohdalla itse Player-luokka eikä PlayerActions-luokka.

Jokaisen luokan pohja luokka on ”Stats” josta löytyvät statistiikat ks. ominaisuudelle. Esimerkiksi Player-luokan kohdalla PlayerStats-luokkaan on määritelty esimerkiksi pelaajan kävelynopeus ja kestävyys. Nämä statistiikat ovat arvoja, joita muokataan pelin aikana, kun pelaaja saa erilaisia päivityksiä hahmoonsa. Kun Stats luokkaa käyttää yksinään komponenttina, voidaan siitä tehdä erilaisia pelaajapäivitys valmis objekteja.

Ominaisuuteen liittyvät toiminnallisuudet, kuten esimerkiksi Player-luokan kohdalla liikkuminen, ovat toteutettu luokan vastaavaan ”Actions” luokkaan. Näiden luokkien alle on koostettu kaikki tarvittavat toiminnot ja niiden käyttämät aputoiminnot. Nämä luokat eivät pääasiassa ota yhteyttä muihin luokkiin ja toimivat listana, josta ylempi luokka voi käyttää tarvittavia toimintoja.

Jokaisen ominaisuuden pääluokka kuten esim. Player-luokka, on vastuussa vastaavan ominaisuuden toiminnasta. Näillä luokilla on tila (state), tila suunnittelumallin mukaan. Luokan tila määrittää mitä toimintoja käytetään ja milloinkin. Controller-luokat muokkaavat ja ohjaavat suoraan näitä luokkia.

Näistä luokista Shooting ja Melee ovat universaaleja molemmille PlayerControllerille ja EnemyControllerille. Player ja Enemy luokissa on monia samankaltaisuuksia, mutta niitä erottaa se miten tietyt asiat kuten liikkuminen on toteutettu ja siksi niiden täytyy olla kaksi erilaista luokkaa. Näitä edellisiä luokkia käyttämällä vihollisobjekteista voidaan rakentaa paljon erilaisia versioita.

5. Pohdinta

Tämä opinnäytetyö oli monivaiheinen ja opettava kokemus. Projektin laajuus muuttui työn aikana moneen otteeseen, koska monet aluksi yksikertaiselta vaikuttavat asiat osoittautuivat monimutkaisemmiksi tarkastellessa niitä ohjelmistoarkkitehtuurin näkökulmasta. Käytin moniin toiminnaltaan yksinkertaisien ominaisuuksien toteuttamiseen paljon enemmän aikaa kuin oli aluksi tarkoitus, koska halusin kehittää koodista paremman arkkitehtuurisesti.

Tämän opinnäytetyön tavoitteena oli toteuttaa mobiilipeli, jonka tulisi täyttää seuraavat tärkeimmät kriteerit: olla arkkitehtuurisesti selkeä ja olla helposti jatkokehitettävä. Lisäksi tavoitteenani oli oppia lisää ohjelmistoarkkitehtuurista ja peli- ja ohjelmistokehityksestä itsessään.

Aloittaessani tätä opinnäytetyötä tiesin ohjelmistoarkkitehtuurista vain perusteet. Minulla oli kuitenkin jo alussa vahva kiinnostus aiheeseen, joka vain kasvoi opiskellessani lähdekirjallisuutta. Lähteistäni haluan vielä nostaa esille Nystromin kirjan Game Programming Patterns, joka nerokkaan kirjoitustyylinsä lisäksi sopi aihealueeltaan täydellisesti tämän projektin tarpeisiin. Opin paljon hyvän arkkitehtuurin periaatteista kuten sidonnaisuudesta ja koodin modulaarisuudesta. Opin myös erilaisista suunnittelumalleista, jotka aiheena osoittautuivat itselleni helpoiten omaksuttavaksi ja käytännönläheisimmäksi osa-alueeksi ohjelmistoarkkitehtuurissa. Tutkin paljon erilaisia suunnittelumalleja ja sovelsin niitä koodissani. Joitain malleja käytin suoraan ja joitain muokkasin enemmän.

Ohjelmistoarkkitehtuurin lisäksi opin projektin aikana myös lisää pelikehityksestä. Pelin monet mekaniikat eivät ole teknisesti monimutkaisia, mutta ne on toteutettu hyvin. Pelin arkkitehtuuri kehittyi iteratiivisesti samalla kun koodin organisointitaitoni kehittyivät projektin aikana. Ajattelutapani koodin suunnittelusta myös kehittyi. Aloin rakentamaan koodia sotkuisten jättimetodien sijaan pienistä selkeistä yksittäisistä pikkumetodeista. Opin myös kirjoittamaan metodit niin että ne olisivat paremmin saavutettavia muualta koodista.

Pelin lähdekoodi on toteutettu modulaarisesti, mikä tukee projektin tavoitetta olla helposti jatkokehitettävissä. Koodin modulaaristen osien avulla peliin voidaan kehittää helposti lisää sisältöä hienosäätämällä ja yhdistelemällä eri elementtejä. Koodi on toteutettu siten että se on sujuvasti muokattavissa myös ilman ohjelmointitaustaa. Tärkeimmät elementit pelissä pelattavan sisällön tuottamisen kannalta on toteutettu siten, että niiden muokkaus onnistuu Unityn editorin kautta. Pelin koodi on myös valmis mahdollisten animaatioiden ja ääniefektien implementointiin, tila-suunnittelumallin käytön ansiosta. Koodin päätoimintaa ei tarvitse muokata tämän kaltaisen sisällön lisäämiseksi.

Opin myös projektin aikana paljon mitä ei kannata tehdä. Versiohallinta ja dokumentaatio jäivät sivuosaan ja niitä ei työssä toteutettu kunnolla. Pidin projektin aikana hyvin niukkaa päiväkirjaan projektin tapahtumista. Pelin arkkitehtuurin dokumentaatio ja muut jatkokehitystä auttavat dokumentit kuten käyttöopas jäivät myös toteuttamatta. Jos tekisin projektin uudestaan, kiinnittäisin näihin osioihin enemmän huomiota. Vaikka projektissa tuotettu peli ei ole näennäisesti monimutkainen, osoittautui se sille varattuun aikaan nähden jokseenkin vaikeaksi. Jos olisin valinnut toteutettavaksi yksinkertaisemman pelin, olisin pystynyt käyttämään enemmän aikaa arkkitehtuurin ja dokumentaation hiomiseen.

Kuitenkin kokonaisuutena koen, että projektin tavoitteissa onnistuttiin. Minulla on nyt paljon enemmän kokemusta ja tietoa ohjelmistoarkkitehtuurista kuin aloittaessani. Peli toimii kokonaisuutena ja se on arkkitehtuurisien rakenteidensa ansiosta helposti jatkokehitettävissä. Kuitenkin pelin toteutuksen ja koodikannan dokumentaatio on niukkaa, joka laskee tuotoksen arvoa. Projektin tuloksena syntynyt peli, sen kehitystä kuvaava sprinttilogi ja teoriaosuudessa käsitellyt konseptit tuovat arvoa mahdollisesti itseni kanssa samankaltaisessa tilanteessa oleville ohjelmistokehittäjille.

6. Lähteet

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (1994) Design Patterns Elements of Reusable Object-Oriented Software. Luettavissa http://www.javier8a.com/itc/bd1/articulo.pdf. Luettu 13.11.2023.

Unity (2023) Game Development Terms. Luettavissa https://unity.com/how-to/beginner/game-development-terms. Luettu 13.11.2023.

Jason W. Bay (2023) Game Industry Career Guide. Luettavissa https://www.gameindustrycareerguide.com/video-game-development-terms-glossary/#E. Luettu 13.11.2023.

YoYo Games. (2023) GameMaker Manual. Luettavissa https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Game_Input/Game_Input.htm Luettu 16.11.2023.

Robert Nystrom. (2014). Game Programming Patterns. Luettavissa https://gameprogrammingpatterns.com/sample.pdf. Luettu 13.11.2023.

Andrew Zola (2022) Instance. Luettavissa https://www.techtarget.com/whatis/definition/instance. Luettu 13.11.2023.

Kai Koskimies, Tommi Mikkonen. (2005). Ohjelmistoarkkitehtuurit. Luettu 13.11.2023.

José P. Zagal, Roger Altizer. (2015) Placeholder Content in Game Development: Benefits and Challenges. Luettavissa https://my.eng.utah.edu/~zagal/Papers/zagal-altizer-placeholdercontentin-games.pdf Luettu 13.11.2023.

Michael Kircher, Prashant Jain. (2002) Pooling. Luettavissa http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf. Luettu 13.11.2023.

Ken Schwaber, Jeff Sutherland. (2020) Scrum-opas Scrumin määritelmä ja pelisäännöt. Luettavissa https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Finnish.pdf. Luettu 13.11.2023.

Len Bass, Paul Clements, Rick Kazman. (2013) Software Architecture in Practice. Luettavissa https://edisciplinas.usp.br/pluginfile.php/5922722/mod_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf. Luettu 13.11.2023.

Unity User Manual (2022). Luettavissa https://docs.unity3d.com/Manual/GameObjects.html. Luettu 13.11.2023.

Liitteet

Liite 1. Linkki projektin GitHub-repositorioon

Pelin kaikki tiedostot https://github.com/henkkarossi/EternalWarrior

Liite 2. Linkki ladattavan pelin sivulle

Pelattava Android build ladattavissa osoitteesta https://github.com/henkkarossi/EternalWarrior/blob/master/ShootingGame03.apk

Liite 3. Linkki videoon pelistä toiminnassa

Youtube-linkki pelivideoon https://youtu.be/tWC3UFKAkyA