Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

JUHO-VILLE METTÄNEN

Pelinkehitys Unity-pelimoottorilla suunnittelusta toteutukseen

TIETOJENKÄSITTELYN TUTKINTO-OHJELMA
2025

TIIVISTELMÄ

Mettänen Juho-Ville: Pelinkehitys Unity-pelimoottorilla suunnittelusta toteutukseen
Opinnäytetyö, AMK
Tietojenkäsittelyn tutkinto-ohjelma
Kesäkuu 2025
Sivumäärä: 58

Tämän opinnäytetyön tavoitteena oli tutustua Unity-pelimoottoriin ja syventyä pelikehityksen prosessiin yksittäisen kehittäjän näkökulmasta. Työn taustalla oli kiinnostus pelinkehitykseen sekä halu oppia käytännön taitoja pelien suunnittelussa ja toteutuksessa. Työssä suunniteltiin ja kehitettiin yksinkertainen pelidemo, jonka avulla testattiin Unityn ominaisuuksia ja kehitystyön vaiheita.

Pelidemon toteutus eteni vaiheittain, alkaen ideoinnista ja pelimekaniikan suunnittelusta, edeten grafiikan, ohjelmoinnin ja testauksen kautta toimivaan kokonaisuuteen. Menetelmänä käytettiin iteratiivista kehitysprosessia, jossa peliä kehitettiin ja paranneltiin vaiheittain testauksesta saatujen havaintojen perusteella.

Tuloksena syntyi pelidemo, joka toimii suunnitellulla tavalla ja havainnollistaa yksittäisen kehittäjän mahdollisuuksia toteuttaa toimiva peliprojekti Unityn avulla. Johtopäätöksenä voidaan todeta, että Unity tarjoaa monipuoliset työkalut pelinkehitykseen ja mahdollistaa tehokkaan työnkulun myös pienille projekteille.

Avainsanat: tietokonepelit, pelisuunnittelu, peliohjelmointi, peliala, peligrafiikka

ABSTRACT

Mettänen, Juho-Ville: Game development with Unity Engine from design to implementation
Bachelor's thesis
Business Information Systems
June 2025
Number of pages: 58

The goal of this thesis was to get acquainted with the Unity game engine and to understand game development from the perspective of a solo developer. The motivation for this work was an interest in game development in general and a desire to learn practical skills in game design and implementation. During this work, a simple video game demo was designed and developed to test out the capabilities of Unity game engine and the steps needed to make it happen.

The implementation of the demo progressed in stages from ideation and game mechanics design, progressing through graphics, programming and testing the functionality. The method used in this work was an iterative development process, where the game was developed and improved step by step based on observations during testing.

The result is a game demo that works as intended and illustrates the possibilities for solo developers to create functional games using Unity. In conclusion, it can be stated that Unity offers versatile tools for game development and enables an efficient workflow even for small projects.

Keywords: computer games, game design, game programming, game sector, game graphics

SISÄLLYS

TERMIT JA LYHENTEET

DLC	Ladattava lisäsisältö. Pelin jälkeen julkaistava lisäpaketti, joka voi sisältää esimerkiksi hahmoja tai uusia tasoja
DOTS	Unityn ohjelmointimalli, joka perustuu datalähtöiseen suunnitteluun. Sen avulla voidaan käsitellä suuria määriä pelin objekteja tehokkaasti rinnakkaisesti, parantaen suorituskykyä erityisesti suurissa peleissä. (eng. Data-Oriented Technology Stack)
Dynaaminen	Muuttuva tai reaaliaikaisesti päivittyvä
Frame	Yksittäinen kuvajärjestelmän piirto- tai päivityskerta. Pelissä frame tarkoittaa yhtä ruutua, joka näytetään näytöllä. Pelin sujuvuus mitataan usein frameina sekunnissa (FPS, frames per second)
HP	Hahmon tai objektin elinvoiman mittari, joka pienenee vahinkojen seurauksena ja nollaan tultua hahmo yleensä tuhoutuu
Integroida	Liittää saumattomasti toiseen järjestelmään tai osaksi kokonaisuutta. (eng. Integrate)
NPC	Peliin kuuluva hahmo, jota ei ohjaa pelaaja, vaan itse peli tai tekoäly
RTRT	Reaaliaikainen säteenseuranta. Kuvien valonheijastuksia ja -kulkuja lasketaan dynaamisesti joka ruudunpäivityksellä. (eng. Real-Time Ray Tracing)

Skripti	Pelissä käytettävä ohjelmakoodi, joka ohjaa tiettyä toimintaa, kuten hahmon liikkeitä, vihollisten käyttäytymistä tai tapahtumien etenemistä.
Sprite	Yksittäinen kuva, kuten hahmo tai esine
Spritesheet	Yksi kuva, joka sisältää useita spritejä.
SRP	Unityn grafiikkamoottorin arkkitehtuuri, jonka avulla kehittäjä voi itse määrittää, miten peli piirtää grafiikat ruudulle. (eng. Scriptable Render Pipeline)
USD	Yhdysvaltain dollari
2D	Lyhenne sanasta kaksiulotteinen
3D	Lyhenne sanasta kolmiulotteinen

1 JOHDANTO

Tämän opinnäytetyön tavoitteena oli perehtyä Unity-pelimoottorin käyttöön ja pelinkehityksen eri vaiheisiin yksittäisen kehittäjän näkökulmasta. Opinnäytetyön toiminnallinen osuus koostui pelidemosta, jossa yhdistyvät pelisuunnittelu, ohjelmointi, visuaalinen ilme sekä pelimekaniikka. Pelidemon tavoitteena oli luoda toimiva kokonaisuus havainnollistamaan pelinkehityksen keskeisiä osa-alueita.

Opinnäytetyön alussa käsittelen Unityn yleisiä ominaisuuksia ja sen soveltuvuutta pelikehitykseen verrattuna muihin pelimoottoreihin, kuten esimerkiksi Unreal Enginen ja Godotiin. Lisäksi käyn läpi pelimoottorin peruskäyttöä sekä sen tarjoamia mahdollisuuksia pelin rakentamisessa. Esittelen käyttämäni työmenetelmät, kuten projektinhallintaan liittyvät käytännöt sekä graafiseen suunnitteluun käytetyn Aseprite-ohjelman. Lopuksi vielä pohdin omaa onnistumistani projektin suhteen, mitä opin, mitä sain tästä irti ja missä olisi vielä parannettavaa.

2 UNITY

Unity on yksi tunnetuimmista ja suosituimmista pelimoottoreista markkinoilla. Sen on kehittänyt Unity Technologies ja sitä on päivitetty sekä ylläpidetty jo vuodesta 2005. Ensimmäinen versio julkaistiin kesäkuussa vuonna 2005. Tavoitteena oli luoda edullinen pelimoottori aloittelijoille ammattitason työkaluilla. (Haas, 2014, s. 1.)

Unity sisältää useita teknologisia edistysaskeleita, kuten Scriptable Render Pipeline (SRP) -järjestelmän, joka mahdollistaa korkeatasoisten grafiikkojen optimoinnin eri alustoille. Lisäksi DOTS (Data-Oriented Technology Stack) -arkkitehtuurin, joka parantaa suorituskykyä suurissa ja monimutkaisissa projekteissa. Unityn sisäänrakennettu fysiikkamoottori, animaatiojärjestelmä ja AI-työkalut mahdollistavat laajan valikoiman pelimekaniikkoja ja reaaliaikaista simulaatiota.

2.1 Lisenssit ja muut kustannukset

Unitylla on käytössä erilaisia lisenssivaihtoehtoja sekä mahdollisia lisäkustannuksia. Tarjolla on eri tasoisia lisenssejä, jotka määräytyvät käyttäjän liikevaihdon tai rahoituksen perusteella. Lisensseihin ja kustannuksiin tuli kuitenkin muutoksia 01.01.2025. Unity Personal on kaikista yksinkertaisin vaihtoehto, joka on tarkoitettu yksittäiselle henkilölle tai pienelle yritykselle. Se sisältää Unityn editorin ja perustoiminnot. Sen käyttö on täysin ilmaista, kunnes liikevaihto tai rahoitus ylittää 200000 USD vuodessa. Unityn vesileima tulee myös olemaan vaihtoehtoinen jatkossa. (Bromberg, 2024.)

Unity Pro ja Unity Enterprise vaihtoehdot ovat myös saaneet muutoksia hinnoitteluun. Unity Pro koki 8 % korotuksen hinnassa ja se nousi 2200 USD vuodessa yhtä työntekijää kohden. Yritysten, joiden vuosittainen liikevaihto tai rahoitus ylittää 200000 USD on käytettävä Unity Pro -versiota. Unity Enterprise sai 25 % korotuksen hintaansa ja on pakollinen, jos vuosittainen liikevaihto tai rahoitus ylittää 25 miljoonaa USD. (Bromberg, 2024.) Hinnoittelun lisäksi

näissä versioissa on eroavaisuuksia. Ilmainen versio sisältää perustoiminnot, kun taas Pro ja Enterprise -versiot sisältävät esimerkiksi mahdollisuuden tuottaa pelejä konsoleille. Enterprise -tilaus antaa myös rajoitetun pääsyn lähdekoodiin, jonka avulla voidaan muokata moottoria tarpeiden mukaan. (Unity Technologies, n.d.)

2.2 Alustat

Unity tukee useita eri alustoja, joihin sisältyy esimerkiksi yleisimmät konsolit kuten Playstation, Xbox ja Nintendo Switch, mobiililaitteet (Android ja iOS), tietokonealustat (Windows, macOS ja Linux) sekä XR-laitteet kuten Meta Quest ja HoloLens (Unity Technologies, 2025c).

3 UNITYN SOVELTUVUUS PELIKEHITYKSEEN

Unity tarjoaa mahdollisuuden korkean tason pelikehitykseen, joka tukee huippuluokan grafiikoita, korkealaatuista ääntä ja sulavaa pelattavuutta, kuten suurien pelistudioiden tuottamissa peleissä. Unityn monipuolinen työkalupakki tekee siitä erinomaisen valinnan sekä 2D- että 3D-pelikehitykseen tarjoamalla kehittäjille lukuisia etuja ja resursseja pelien luomiseen. Helppokäyttöisten työkalujen avulla editoria voi muokata omien tarpeiden mukaisesti. Sisäänrakennettu animaatiojärjestelmä mahdollistaa luonnollisten animaatioiden tekemisen nopeasti. Lisäksi monialustainen tuki mahdollistaa pelien julkaisun eri alustoille ja laajan valikoiman valmiita 2D- ja 3D-resursseja eri tarkoituksiin. Näillä voi nopeuttaa työn tekemistä ja säästää aikaa, jota voidaan hyödyntää pelikehityksen muissa osissa. Kaiken kaikkiaan Unity on todella käyttäjäystävällinen alusta tarkoittaen, että sekä aloittelijat että kokeneemmat kehittäjät pystyvät työskentelemään tehokkaasti. (Martynenko, 2022.)

3.1 Unityn vahvuudet ja käyttötarkoitukset

Unitylla on useita vahvuuksia, jotka tekevät siitä erinomaisen pelimoottorin pelikehitykseen. Yksi isoimmista tekijöistä on Unityssä ohjelmointikielenä käytettävä *C#*. Ohjelmointikielen etuna on sen suhteellisen helppo ja tasainen oppimiskäyrä, joka laskee aloittelijoiden kynnystä lähteä opettelemaan pelikehitystä sekä kannustaa heitä jatkuvaan oppimiseen ja kehittymiseen. Unreal Enginessä käytettävän C++ kielen oppimiskäyrä on paljon jyrkempi verrattuna C# -ohjelmointikieleen. Unity antaa erinomaista tukea aloitteleville kehittäjille ja auttaa saamaan vaikuttavia tuloksia. Kaksi hyvää esimerkkiä Unitylla kehitetyistä peleistä on Cuphead ja Ori and the Will of the Wisp, jotka molemmat ovat 2D-pelejä. Unity on myös yksi yleisimmin käytetyistä pelimoottoreista mobiilialustan peleille, sillä sen monipuoliset ominaisuudet mahdollistavat helpon siirtämisen eri jakelualustoille ja laitteille. (Martynenko, 2022.)

3.2 Muut pelimoottorit

Pelikehitykseen löytyy Unityn lisäksi useita muita pelimoottoreita, joista suosituimpia vaihtoehtoja ovat Godot ja Unreal Engine, jotka tarjoavat erilaisia ominaisuuksia eri kehittäjäryhmille ja pelityypeille. Jokaisella pelimoottorilla on omat vahvuutensa ja heikkoutensa.

Godot on avoimen lähdekoodin pelimoottori, joka on noussut nopeasti Unityn kilpailijaksi pelikehityksessä erityisesti indie -kehittäjien keskuudessa. Sen käyttö on täysin ilmaista lukuun ottamatta maksullisia liitännäisiä ja resursseja. Godot on kevyt ja tehokas tarkoittaen sitä, että se käyttää vähemmän resursseja kuin Unity ja Unreal sekä myös omaa GDScript -kieltä, joka muistuttaa todella paljon Python-ohjelmointikieltä. Kuten Python myös GDScript on aloittelija ystävällinen ohjelmointikieli. Godotin suurimpana vahvuutena on sen 2D-pelikehityksen tuki. (RocketBrush Studio, 2024a.)

Godot pelimoottorilla on mahdollista kehittää myös 3D-pelejä kuten muillakin pelimoottoreilla, mutta sen soveltuvuus ei ole samalla tasolla kuin Unrealin ja Unityn. Godot on suhteellisen uusi pelimoottori, joten sillä ei ole vielä suurta käyttäjänkuntaa. Tämän vuoksi verkossa ei ole paljoa tietoa, videoita tai tutoriaaleja pelimoottorin käyttöön liittyen. Moottori ei ole vielä noussut suosioon suurten yritysten keskuudessa, joka on vaikuttanut sen käyttäjämäärään negatiivisesti. (RocketBrush Studio, 2024a.)

Unreal Engine tunnetaan sen laadukkaasta visuaalisesta näkymästään ja Real-Time Ray Tracing -ominaisuudestaan, jonka avulla pystytään rakentamaan eläviä ympäristöjä. Kuten Unityn Visual Scripting, sisältää myös Unreal oman Blueprint -järjestelmänsä. Tämän järjestelmän avulla voidaan rakentaa monimutkaisia pelimekaniikkoja ilman ohjelmointia. Peli pyörii sujuvasti, vaikka siinä olisi paljon korkealaatuisia resursseja ja tehosteita esillä, sillä Unreal Engine on optimoitu suorituskykyä varten. Jos korkealaatuiset grafiikat ovat pelin ensisijainen tavoite, on Unreal Engine oikea vaihtoehto. Pelimoottorin ominaisuudet tekevät siitä parhaan vaihtoehdon peleille, joiden tavoitteena on näyttää visuaalisesti hyvältä. Epic Games tarjoaa myös pääsyn moottorin

lähdekoodiin ilmaiseksi, joka mahdollistaa pelimoottorin muokkauksen omien tarpeiden mukaisesti. Unreal Enginen ympärillä pyörii myös suuri aktiivinen yhteisö ja tarjolla on paljon opetusvideoita, tutoriaaleja ja resursseja eri tarpeisiin. Unreal Enginessä on kuitenkin todella jyrkkä opettelemisen käyrä. Vaikka pelimoottori onki todella tehokas ja edistynyt, on se myös monimutkaisempi käyttää ja opetella kuin Unity. Tämä voi olla este aloittelijoille ja pienille yrityksille. Sen lisäksi itse pelimoottorin pyörittäminen vaatii tehokkaan tietokoneen, jotta se pyörii sujuvasti. Osaa ominaisuuksista, kuten esimerkiksi säteenseurantaa ei ole mahdollista edes käyttää, mikäli tietokone ei täytä tiettyjä järjestelmävaatimuksia. (RocketBrush Studio, 2024b.)

4 UNITYN SISÄLTÖEDITORI

4.1 Unityn käyttöönotto: järjestelmävaatimukset

Unity pelimoottorin voi ladata Unityn kotisivuilta ilmaiseksi. Sivulta löytyy myös opiskelijoille ilmainen versio, joka sisältää 20 % alennuksen Asset Storen resursseista (Unity Technologies, n.d.) Unityn käyttöönotto edellyttää, että käyttäjän tietokone täyttää tietyt järjestelmävaatimukset. Vaatimukset voivat kuitenkin vaihdella Unityn eri versioiden, projektin koon ja kehitysalustojen mukaan. Unity on suhteellisen kevyt pelimoottori verrattuna Unreal Engineen, mutta suorituskyky ja käyttökokemus paranevat huomattavasti tehokkaammalla tietokoneella. Kehitysympäristön valintaan vaikuttavat pelityyppi, grafiikat ja mahdollinen VR/AR-tuki. (Unity Technologies, 2025c.)

Taulukko 1. Unityn käyttöönoton järjestelmävaatimukset (Unity Technologies, 2025c)

Vaatimukset	Unity	Työssä käytetty tietokone
Käyttöjärjestelmä	Windows 10 versio 21H1 tai uudempi	Windows 10 22H2
Prosessori	x86-64-arkkitehtuurin prosessori, jossa on SSE2-käskykannan tuki	13. sukupolven Intel® Core™ i5-13600K, 3500 MHz, 14 ydintä
Näytönohjain	DX10, DX11, DX12 Tai Vulkania tukeva GPU	NVIDIA GeForce RTX 3070 Ti
Muisti	Riippuu projektin koosta suurimmaksi osaksi.	16GB

4.2 Unityn liitännäiset ja Asset Store

Liitännäiset ovat erillisiä ohjelmistoja, jotka integroituvat Unityyn ja lisäävät uusia ominaisuuksia tai parantavat olemassa olevia. Esimerkiksi liitännäisillä voidaan lisätä VR- tai AR-tuki (Virtual- ja Augmented reality) lataamalla niihin tarvittavat ohjelmistot. Latauksen yhteydessä Unityyn asennetaan valmiit työkalut ja toiminnot, jotka mahdollistavat näiden ominaisuuksien käyttämisen. Liitännäiset voivat myös parantaa pelin grafiikkaa, kuten optimoimalla valaistusta tai lisäämällä realistisia varjoefektejä.

Liitännäiset ovat keskeinen osa Unity-pelimoottoria, koska ne tuovat tärkeitä ominaisuuksia, joita moottorissa ei välttämättä alun perin ole. Tämä säästää kehittäjiltä aikaa ja vaivannäköä, koska heidän ei tarvitse muokata moottoria tarpeidensa mukaan, vaan voivat suoraan hyödyntää valmiita liitännäisiä, jotka tukevat haluttuja toimintoja.

4.2.1 Asset Store

Unityssa, kuten monessa muussakin pelimoottorissa, on erillinen kauppa resursseille, joka tunnetaan nimellä Asset Store. Kaupasta voi ladata tarvittavat 2D-mallit ja -materiaalit, animaatiot, äänitehosteet, mutta myös paljon muuta. Tämä kaupallinen ekosysteemi tarjoaa laajan valikoiman valmiita resursseja, jotka voivat nopeuttaa pelinkehitystä ja parantaa projektin laatua.

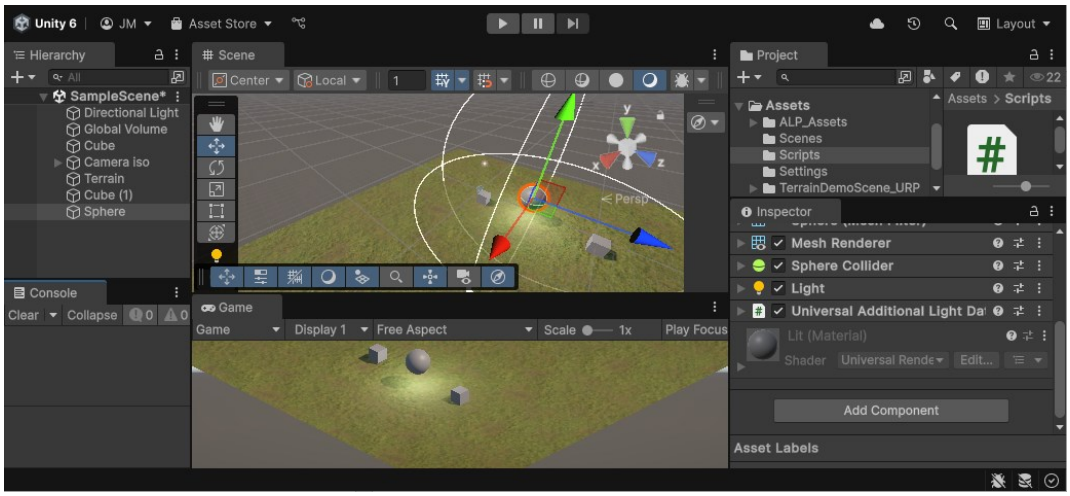
Asset Store:n avulla kehittäjät voivat valita sopivia resursseja, jotka täyttävät pelin tarpeet ja visuaaliset vaatimukset, ilman että heidän täytyy aloittaa jokaista elementtiä alusta. Tämä mahdollistaa nopeamman kehitysprosessin ja paremman keskittymisen pelin muihin osiin, kuten pelisuunnitteluun ja tarinankerrontaan.

4.3 Sisällönhallinta ja editori

Unityn editorissa on monipuoliset työkalut projektin hallintaan. Sisällönhallinta kattaa kaiken projektin tiedostojen, kuten 3D- ja 2D-mallien, tekstuurien, äänien ja skriptien organisoinnista niiden käyttöön ja optimointiin. Kaikki projektin tiedostot säilytetään Assets-kansiossa, joka löytyy Project-ikkunasta. Inspector-ikkuna näyttää valitun GameObjectin tiedot ja antaa mahdollisuuden muokata sen ominaisuuksia ja komponentteja.

Scene on näkymä, jonne pelin sisältö luodaan ja missä sitä muokataan. Se voi sisältää esimerkiksi hahmoja, ympäristöjä, pelitason rakenteita ja käyttöliittymän. Yhteen sceneen voi rakentaa kokonaisen pelin, mutta peli on todennäköisesti yksinkertainen. Monimutkaisemmissa peleissä jokainen taso voi olla rakennettuna omaan sceneensä. (Unity Technologies, 2025b.)

Oletuksena vasemmassa yläkulmassa sijaitsee Hierarchy-ikkuna, jossa näkyvät kaikki scenessä olevat GameObjectit. Niitä voi lisätä peliin napsauttamalla hiiren oikealla painikkeella Hierarchy-ikkunassa ja valitsemalla halutun objektin. Oikeassa reunassa sijaitsee Project-ikkuna, joka sisältää kaikki pelissä tarvittavat tiedostot ja resurssit. Se toimii samalla tavoin kuin tietokoneen tiedostojenhallinta. Hierarchy-ikkuna näyttää vain yhden scenen objektit, eikä koko projektin tiedostorakennetta. Kuvassa 1 näkyy Unity editorin perusnäkymä.

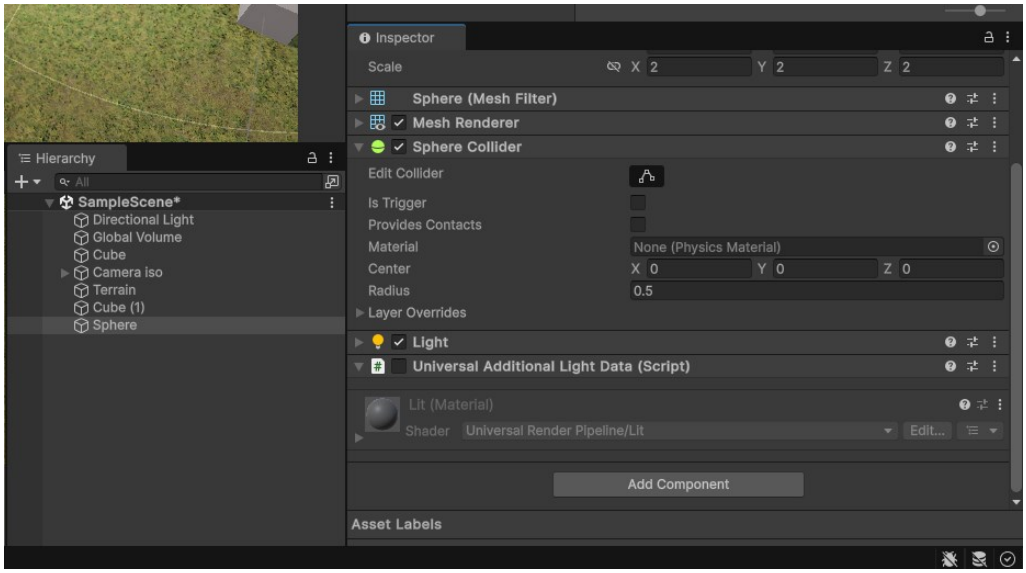


Kuva 1. Kuvankaappaus Unityn editorista.

4.4 GameObjects ja Components

GameObject on kaikista oleellisin osa Unityn editoria. Jokainen pelissä oleva asia tai esine on GameObject, kuten hahmot, kerättävät esineet, valot, kamerat ja erikoistehosteet. GameObject itsessään ei kuitenkaan ole toiminnallinen ilman lisäominaisuuksia. Jotta objekti voi toimia esimerkiksi hahmona, ympäristönä tai erikoistehosteena, sille on määriteltävä tarvittavat komponentit ja ominaisuudet. GameObjectit ovat kaiken perusta, mutta sellaisenaan ne eivät saa juuri mitään aikaiseksi. Sen sijaan ne toimivat kontteina komponenteille, jotka määrittävät objektin toiminnallisuuden. (Unity Technologies, 2025a.)

Tilemap on eräänlainen GameObject, jonka tarkoitus on auttaa 2D-pelikentän rakentamisessa ruudukoiden avulla. Siihen voi maalata Tile paletista alueita (tiles), kuten maa ja seinät. Se toimii Grid-objektin alla ja tukee eri asettelutyyppejä, kuten suorakulmainen tai isometrinen. Se helpottaa kenttien visuaalista suunnittelua ja toimii yhdessä komponenttien kanssa, kuten Tilemap Collider. (Unity Technologies, 2023.) Komponentit vastaavat GameObjectin toiminnallisuudesta oli se sitten valo, kamera tai paikallaan oleva laatikko. Unity sisältää useita valmiita komponentteja ja niitä on mahdollista tehdä itse käyttämällä Unity Scripting API:a, Esimerkkinä kuvassa 2 GameObjectista Sphere voidaan tehdä valonlähde liittämällä siihen valokomponentti. (Unity Technologies, 2025a.)



Kuva 2. Kuvakaappaus komponentin liittämisestä GameObjectille.

4.5 Visual Scripting

Unityssä on käytössä myös Visual Scripting ominaisuus, joka mahdollistaa skriptien luomisen ilman perinteistä koodausta. Tämä ominaisuus perustuu solupohjaiseen koodaus menetelmään ja sen avulla voidaan rakentaa syy-seuraus tapahtumia esimerkiksi ragdoll-effect. Pelaajan osuessa seinään tapahtuma X toteutuu, jonka seurauksena pelaajan hahmo kaatuu. Vaikka tämä menetelmä ei tarjoa samanlaista vapautta ja suorituskykyä kuin tapahtuman ohjelmointi itse, auttaa se kehittämään taitoja ja ymmärtämään kehitysprosessia paremmin. (Martynenko, 2022.)

5 TYÖSKENTELYMENETELMÄT JA TYÖVÄLINEET

5.1 Projektinhallintamenetelmät

Projektihallintamenetelmät ovat järjestelmällisiä tapoja suunnitella, hallita ja seurata projektin etenemistä. Niiden avulla varmistetaan, että projekti etenee aikataulun mukaisesti, pysyy budjetissa ja saavuttaa asetetut tavoitteet. Jotta projekti onnistuu, on tärkeää käyttää selkeää työskentelytapaa ja suunnitella huolellisesti riippumatta projektin koosta tai tiimin vahvuudesta. Pelikehityksessä on tärkeä valita sopiva hallintamenetelmä, joka tukee tiimin tai yksittäisen kehittäjän työnkulkua. (Estevez, 2015.)

Monet yrittäjät jättävät projektin kesken kokiessaan työmäärän liian suureksi ja ymmärtäessään sen laajuuden. Tämä on erityisen yleistä yksin työskentelevillä pelinkehittäjillä, jolloin vastuu kasaantuu vain yhden henkilön harteille. Tilannetta voi helpottaa pilkkomalla suuren projektin pienempiin osiin, kuten tiettyihin vaiheisiin tai virstanpylväisiin sekä jakamalla ne vielä yksityiskohtaisempiin tehtäviin. Näin työ tuntuu hallittavammalta ja jokainen tehtävä on selkeämpi, nopeammin suoritettavissa ja saavutettavissa. Keskittymällä yhteen asiaan kerrallaan voidaan välttää turhaa stressiä ja seurata omaa edistymistä konkreettisemmin. (Estevez, 2015.)

5.1.1 Vesiputousmalli

Vesiputousmalli (Waterfall) sisältää tarkat aikataulut ja budjettiarviot projektiin. Mallissa ei myöskään tehdä asioita projektin aikataulutuksen ulkopuolella vaan tehtävät suoritetaan siinä järjestyksessä, mihin ne on etukäteen asetettu. Tämä tuo selkeyttä ja järjestelmällisyyttä etenkin silloin, kun projektin vaatimukset tiedetään hyvin jo alussa. Vesiputousmallin vaiheet etenevät loogisesti seuraavalla tavalla:

- määritellään vaatimukset
- suunnitellaan

- toteutetaan
- testataan
- julkaistaan valmis tuote.

Jokainen vaihe on edellytys seuraavalle, eikä edelliseen palata, ellei ole aivan pakko. Tämän vuoksi malli edellyttää tarkkaa ennakkosuunnittelua ja lopullisen tuotteen on oltava jo varhaisessa vaiheessa tiedossa. (Atlassian, n.d.)

5.1.2 Ketterät menetelmät Kanban ja Scrum

Tyypillisesti pelikehityksessä käytetään ketteriä menetelmiä, kuten Kanban tai Scrum (Estevez, 2015). Ketterien menetelmien käyttö pelikehityksessä tuo monia etuja. Ne tehostavat työskentelyä, sillä lyhyet ja selkeät työvaiheet eli sprintit auttavat tiimejä havaitsemaan ja korjaamaan ongelmia nopeasti. Lisäksi menetelmien joustavuus mahdollistaa uusien ideoiden ja teknologioiden hyödyntämisen projektin aikana, mikä varmistaa, että lopullinen peli on ajan tasalla ja mahdollisimman kehittynyt. (Streamline Studios, 2024.)

Kanban on kevyt ja joustava projektinhallintamenetelmä, jossa korostuu työn visualisointi, tehtävien määrän rajoittaminen ja jatkuva kehittäminen. Se ei ole yhtä tarkasti määritelty kuin Scrum, joten sen käyttöönotto on helpompaa. Työskentelyä ohjataan Kanban-taululla, jossa tehtävät etenevät vaiheittain sarakkeesta toiseen. Jokaisella vaiheella on rajattu määrä samanaikaisia tehtäviä (Work in Progress -rajoite), mikä auttaa tunnistamaan pullonkaulat ja tehostamaan työn kulkua. (Anders Innovations, 2025.)

Scrum -menetelmän perusrakenne on yksinkertainen: Scrum -tiimi koostuu tuoteomistajasta, Scrum Masterista ja kehittäjistä. Jokaisella roolilla on selkeät vastuualueet. Tiimi osallistuu viiteen säännölliseen tapahtumaan (kuten sprinteihin ja katselmointeihin) ja tuottaa kolme keskeistä työvälinettä, jotka auttavat kehitystyön seurannassa. Scrum perustuu empiiriseen lähestymistapaan, eli päätöksiä tehdään havainnoinnin, kokemuksen ja kokeilujen perusteella. Kolme keskeistä peruspilaria ovat läpinäkyvyys, tarkastelu ja sopeutuminen.

Työ etenee vaiheittain ja jokaisesta vaiheesta opitaan, minkä pohjalta toimintaa voidaan kehittää edelleen. Luottamus on olennainen osa tiimin toimivuutta. Ilman sitä yhteistyö voi olla vaikea ja työn eteneminen voi hidastua. Scrumissa korostetaan viittä arvoa: rohkeus, keskittyminen, sitoutuminen, kunnioitus ja avoimuus. Nämä arvot tukevat avointa ja rakentavaa ilmapiiriä, jossa jatkuva kokeilu ja kehittyminen on mahdollista. (Scrum, n.d.)

5.1.3 Sprintit

Sprintti on lyhyt ja ennalta rajattu kehitysjakso (yleensä 1–4 viikkoa), jossa tiimi toteuttaa valitun osan projektista. Se on keskeinen osa Scrum -menetelmää, joka sisältyy ketteriin menetelmiin. Sprinttien avulla suuri kokonaisuus jaetaan hallittaviin osiin, mikä mahdollistaa nopeamman kehityksen, jatkuvan palautteen ja joustavan reagoinnin muutoksiin. Jokainen sprintti sisältää suunnittelun, toteutuksen, testauksen ja katselmoinnin sekä lopuksi arvioidaan tulokset ja parannetaan prosessia seuraavaa sprinttiä varten. (Rehkopf, n.d.)

5.2 Asesprite

Aseprite on monipuolinen pikseligrafiikkatyökalu, joka on suunniteltu erityisesti videopelien grafiikan luomiseen. Se mahdollistaa hahmojen, ympäristöjen ja erilaisten efektien piirtämisen sekä animaatioiden tekemiseen pikselitarkasti. Aseprite tarjoaa monipuoliset työkalut pikselitaiteen luomiseen ja muokkaamiseen. Kynätyökalu, täyttötyökalu, gradientit, tasot ja erilaiset sekoitustilat tekevät grafiikan suunnittelusta joustavaa. Lisäksi ohjelma tukee spritesheettejä ja frame-pohjaista animointia, mikä helpottaa hahmojen ja pelimaailman dynaamisten elementtien luomista.

Aseprite tallentaa projektit omaan aseprite tai ase muotoihin, säilyttäen kaikki tiedot kuten väri, tasot ja animaatiokehykset. Valmiit grafiikat voi muuntaa yleisiin muotoihin, kuten .png tai .gif, mutta alkuperäinen tiedosto kannattaa säilyttää muokkaamista varten. Käyttäessä Asepritellä tehtyjä grafiikkoja

julkaistavassa materiaalissa, tulee käyttää esimerkiksi .png tiedostotyyppiä. (Aseprite, n.d.)

6 PELIKEHITYS VAIHEITTAIN

6.1 Pelikonsepti

Pelikehitys lähtee aina ideasta. Uusi peli-idea voi syntyä milloin tahansa, se voi nousta esiin esimerkiksi yksittäisestä ajatuksesta, kokemuksesta tai henkilökohtaisista mielenkiinnon kohteista. Uuden peli-idean tullessa mieleen on tärkeää muokata alkuperäistä ideaa tarkemmaksi ja kehittää siitä toimiva kokonaisuus. Suunnitteluvaiheessa määritellään pelin perusperiaatteet, tavoitteet ja visio, jotka ohjaavat koko kehitysprosessia.

Pelikonsepti tiivistää pelin ytimen niin, että sen tarkoitus, sisältö ja kohdeyleisö käyvät selvästi ilmi. Sen tehtävänä on kertoa mikä tekee pelistä kiinnostavan ja millä tavalla se erottuu muista. Suunnittelijan vastuulla on varmistaa, että pelin idea pysyy selkeänä ja johdonmukaisena koko kehityksen ajan sekä että se on dokumentoitu tavalla, jonka jokainen tiimin jäsen ymmärtää. (Kramarzewski & De Nucci, 2018, s. 18.)

6.2 Pelisuunnittelu

Hyvä pelisuunnittelu syntyy, kun sitä tarkastellaan eri suunnista ja näkökulmista. Näitä voi ajatella ikään kuin työkaluina, joilla peliä voi analysoida monipuolisesti. Ne eivät tarjoa valmiita ratkaisuja, vaan herättävät kysymyksiä ja tukevat suunnitteluprosessin hahmottamista eri tasoilla. (Schell, 2008, s. xxvi.) Pelisuunnittelu on yksinkertaistettuna päätösten tekemistä. Pelin rakenne, mekaniikat, tarina sekä visuaalinen tyyli vaativat harkittuja valintoja ja usein koko peli voidaan suunnitella mielessä ennen sen siirtämistä käytännön tasolle. On hyvä tapa opetella kirjoittamaan kaikki ajatukset ja ideat ylös, jotta mitään tärkeää ei unohdu. Näin varmistetaan, että suunnitteluprosessi etenee johdonmukaisesti ja ideat eivät hajoa tai katoa matkan varrella. (Schell, 2008, s. xxiv.)

6.2.1 Tarina ja maailmanrakentaminen

Tarina ja pelimaailma kulkevat käsi kädessä luoden merkityksellisen ja todentuntuisen kokemuksen pelaajalle. Tarina tarjoaa kontekstin ja syyn pelin tapahtumille, kun taas pelimaailma toimii sen näyttämönä. Joissain peleissä, kuten Final Fantasy -sarjassa, tarina on monimutkainen ja pitkäkestoinen, kun taas toiset pelit voivat hyödyntää vain kevyttä narratiivista kehystä. Esimerkiksi shakki voisi olla täysin abstrakti peli, mutta sen taustalla on kuvitteellinen taistelu kahden keskiaikaisen valtakunnan välillä. (Schell, 2008, s. 262–263.)

Pelit erottuvat perinteisestä tarinankerronnasta siinä, että ne mahdollistavat epälineaarisuuden ja pelaajan vaikuttamisen tarinan kulkuun. Tämä haastaa perinteiset tarinan rakenteet ja tekee pelikokemuksesta ainutlaatuisen. Pelaajat eivät kuitenkaan aina tarvitse valmista tarinaa, vaan usein he luovat sen itse. Yksinkertaisissakin peleissä, joissa ei ole sisäänrakennettua juonta, pelaajat saattavat keksiä omia tarinoitaan. Esimerkiksi eräässä noppapelissä lapset innostuivat leikkimään merirosvoja, jotka pelasivat sieluistaan, vaikka peli itsessään oli täysin abstrakti. (Schell, 2008, s. 262–263.)

Maailmanrakennuksella on keskeinen rooli tarinallisen pelikokemuksen luomisessa. Hyvin suunniteltu maailma sisältää historiaa, kulttuuria ja sääntöjä, jotka tekevät siitä uskottavan ja elävän. On tärkeää, että pelin sisäinen logiikka pysyy johdonmukaisena. Kun pelimaailman säännöt on asetettu, tulee niitä noudattaa, ellei tarinassa ole todella painavaa syytä niiden muuttamiseen. Tämä yhtenäisyys mahdollistaa sen, että pelaajat uskovat pelin maailmaan ja pääsevät täysin uppoutumaan siihen. (BIMM University, 2024.) Lopulta tärkeintä ei ole pelkästään tarinan tai pelimekaniikan luominen, vaan vaikuttavan ja mieleenpainuvan kokemuksen tarjoaminen, jossa tarina ja peli sulautuvat yhdeksi kokonaisuudeksi (Schell, 2008, s. 262–263).

6.2.2 Pelimekaniikkojen suunnittelu

Pelimekaniikat ovat pelin toimintalogiikka ja periaatteet, jotka määrittävät miten pelaaja voi toimia pelimaailmassa. Ne kertovat mikä on pelin tavoite, millä

tavoin pelaajat voivat pyrkiä saavuttamaan sen ja mitä seurauksia heidän teoillaan on. (Schell, 2008, s. 41.) Yhdessä pelissä jokin toiminto voi olla keskeinen mekaniikka, kun taas toisessa se voi olla vain koristeellinen lisäys. Olennaista on, että mekaniikka vaikuttaa pelin kulkuun ja sen seuraukset tuntuvat pelaajalle merkityksellisiltä. (Brazie, n.d.)

Pelimekaniikat voidaan jakaa kolmeen pääkategoriaan niiden tarkoituksen mukaisesti: ydinmekaniikat, päämekaniikat ja sivumekaniikat. Hyvin menestyneissä peleissä on yleensä keskitytty pääsijaisesti yhteen ydinmekaniikkaan, jonka ympärille on rakennettu toistuvat haasteet. (Brazie, n.d.) Kaikista tunnetuimmat pelimekaniikat ovat Super Mario Bros pelin ydinmekaniikat hyppiminen ja juokseminen (Kramarzewski & De Nucci, 2018, s. 97). Haasteena on kuitenkin erilaiset esteet, kuten seinät ja liikkuvat viholliset. Sellaisenaan peli ei kuulosta hirveä monimutkaiselta ja ei välttämättä herätä kiinnostusta, mutta sitä varten peliin on luotu pää- ja sivumekaniikkoja rikastuttamaan ydinmekaniikkaa. (Brazie, n.d.)

Päämekaniikat ovat mukana tukemassa ydinmekaniikkaa, mutta niillä ei pyritä tekemään pelistä liian monimutkaista. Esimerkiksi vihollisten tuhoaminen hyppäämällä niiden päälle on keskeinen toiminto, joka yhdistyy pelin ydinmekaniikkaan ja tuo siihen lisäominaisuuden. Sivumekaniikat puolestaan lisäävät peliin syvyyttä ja monipuolisuutta. Ne eivät ole välttämättömiä pelin toiminnan kannalta, mutta niiden lisääminen rikastuttaa pelaajan kokemusta ja tuo peliin uusia haasteita. Esimerkiksi erikoiskohteet tai tasojen erilaiset esteet voivat tuoda lisää mielenkiintoa ja monipuolisuutta pelikokemukseen. (Brazie, n.d.)

6.2.3 Graafinen tyyli ja audiovisuaalinen suunnittelu

Estetiikka on yksi pelisuunnittelun keskeisistä osa-alueista. Jotkut suunnittelijat saattavat pitää sitä vain pinnallisena lisänä, jolla ei ole vaikutusta itse pelimekaniikkaan. Todellisuudessa peli ei koostu pelkästään mekaniikoista, vaan kokonaisvaltaisesta kokemuksesta, jossa myös visuaalinen tyyli ja audiovisuaalinen suunnittelu ovat merkittävässä roolissa. Hyvin toteutettu ulkoasu voi

tehdä pelistä huomattavasti vaikuttavamman ja mieleenpainuvamman. (Schell, 2008, s. 347.)

Tarkastellaan esimerkiksi Amnesia: The Dark Descent -peliä. Vaikka pelin grafiikat ovatkin vanhentuneet ajan myötä, on pelin tunnelma edelleen yhtä vahva. Pimeät käytävät tuntuvat edelleen ahdistavilta, hirviöt karmivilta sekä ympäristöt levottomuutta herättäviltä. Suurin osa tästä on pelin äänisuunnittelun ansiota. (Cuaycong, 2024.) Visuaalinen tyyli ja äänisuunnittelu eivät siis ole vain esteettisiä valintoja, vaan ne voivat myös vahvistaa pelin teemaa ja luoda syvemmän tunteellisen yhteyden pelaajaan. Esimerkiksi tumma ja synkkä visuaalinen tyyli voi tukea kauhu- tai selviytymispelien tunnelmaa, kun taas kirkkaat värit ja iloiset äänet voivat luoda kevyen ja hauskan pelielämyksen.

On helppo keskittyä pelkästään visuaaliseen ilmeeseen mietittäessä pelin estetiikkaa, mutta äänellä on valtava merkitys. Äänipalautteet tuntuvat usein voimakkaammilta kuin visuaaliset efektit ja voivat luoda tunteen kosketuksesta. Eräässä tutkimuksessa pelaajat arvioivat pelin grafiikkaa, mutta heille ei kerrottu, että testin varsinainen ero oli äänenlaadussa. Toinen ryhmä pelasi matalan äänenlaadun versiota ja toinen korkealaatuista versiota. Yllättäen ryhmä, jolla oli parempi ääni, arvioi myös pelin grafiikan paremmaksi, vaikka visuaaliset elementit olivat identtiset molemmissa versioissa. Tämä osoittaa, kuinka tärkeä rooli äänellä on kokonaisvaltaisen pelikokemuksen luomisessa. (Schell, 2008, s. 351.)

6.3 Tuotanto

Tuotantovaihe on pelinkehityksen kiireisin ja laajin osuus, johon osallistuu suurin tiimi. Sen kesto vaihtelee yhdestä neljään vuoteen, mutta suuremmat projektit voivat kestää vielä pidempään. Tämän vaiheen aikana peliä kehitetään ja hiotaan kaikilla osa-alueilla, kuten tarinan, hahmojen, ympäristöjen ja pelin muiden elementtien osalta. (Denisyuk, 2024.) Tässä vaiheessa peli alkaa muotoutua ja herätä eloon. Kehittäjät ja suunnittelijat rakentavat pelimaailmaa, luovat dynaamisia ympäristöjä ja ohjelmoivat niiden toiminnallisuuksia.

Päähenkilöt ja NPC-hahmot (Non-Player Character) suunnitellaan, mallinnetaan ja animoidaan. Ääninäyttelijät äänittävät dialogia ja hienosäätävät repliikkejä, kun taas äänisuunnittelijat tuottavat peliin tarvittavat ääniefektit ja musiikin. Kirjoittajat viimeistelevät käsikirjoituksen sekä vastaavat pienemmistä tekstisisällöistä kuten hahmojen nimistä ja esineiden kuvausteksteistä. (Brazie, n.d.)

Pelikehityksessä on hoidettavana monia eri tehtäviä kuten aiemmin mainitut ohjelmointi, pelisuunnittelu ja äänisuunnittelu. On tärkeää, ettei yritetä tehdä useita asioita samanaikaisesti. Tehtävät kannattaa jakaa eri osiin ja keskittyä tiettyyn osa-alueeseen kerrallaan mahdollisimman pitkän ajan. Usean eri asian tekeminen samanaikaisesti voi johtaa virheisiin ja heikentää keskittymistä. Vaikka tätä ei aina voida täysin välttää, keskeytyksiä kannattaa minimoida. Esimerkiksi siirtymistä ohjelmoinnista piirrostyöhön heti uuden inspiraation tullessa mieleen kannattaa välttää, sillä se voi vaikuttaa työn tehokkuuteen. Tehtävästä toiseen vaihtaminen kesken kaiken aiheuttaa niin sanotun kontekstin vaihto kustannuksen. Se tarkoittaa siirtymistä tehtävästä toiseen, mikä vie aikaa ja vaatii uudelleen keskittymistä. Jokainen siirtymä hidastaa työskentelyä, sillä uuteen tehtävään palaaminen vaatii totuttelua. Tämän vuoksi selkeästi erilliset työskentelyajat eri tehtäville parantavat sekä tuottavuutta että työskentelytapojen hallintaa. On suositeltavaa kokeilla varata omat päivät eri työtehtäville ja seurata, miten se vaikuttaa suoritukseen. (Estevez, 2015.)

6.3.1 Testausvaihe

On tärkeää varmistaa, että peli toimii suunnitellusti. Testaaminen aloitetaan heti, kun pelistä on olemassa pelattava versio. Aluksi testaus vie vain vähän aikaa, mutta kehityksen loppuvaiheessa se vaatii useiden henkilöiden täysipäiväistä työtä. (Nuclino, n.d.) Tässä vaiheessa peli käydään läpi perusteellisesti, jotta voidaan tunnistaa ja korjata mahdolliset virheet, häiriöt, hyväksikäytettävät mekaniikat tai tilanteet, joissa peli jumittuu (Bramble, 2023). Tämän jälkeen korjataan löydetyt ongelmat ja jatketaan tuotantoa seuraavaan vaiheeseen, joka taas testataan uudelleen.

Pelikehityksessä on vaikea ennustaa, kuinka monta kehityskierrosta tarvitaan ennen kuin peli on valmis. Jokainen lisäkierros parantaa peliä hieman, joten parannustyö voisi periaatteessa jatkua loputtomiin. Siksi kehittäjien on tärkeää löytää tasapaino ja varmistaa, että peli hiotaan riittävän hyväksi ennen kuin resurssit, kuten aika ja budjetti loppuvat. Lopulta työ ei koskaan ole täysin valmis, vaan kehitys päättyy pelin ollessa tarpeeksi valmiiksi julkaistavaksi. (Schell, 2008, s. 94.)

6.4 Pelin julkaiseminen

Peliä kannattaa markkinoida ennen sen julkaisemista. Pelin toivelistaukset voivat antaa paljon tarvittua motivaatiota saattaa peli loppuun asti. Tätä varten pelin tulisi olla esiteltävässä kunnossa, jotta siitä voidaan tehdä demo, traileri tai julkaista se Early Access -tyylisesti. Suuremmat ja rahoitusta omaavat studiot esittelevät pelinsä usein tapahtumissa kuten E3 tai State of Play (Bramble, 2023). Myös sosiaalinen media on tärkeä markkinointikanava. Esimerkiksi pelin kehityksestä kertovat kuvat ja videot YouTubessa, Tiktokissa ja Instagramissa voivat herättää kiinnostusta ja luoda yhteisöä pelin ympärille. Pelin julkaisualustalla on suuri merkitys markkinoinnissa ja jälkituotannossa. Esimerkiksi Steam tarjoaa työkalut toivelistojen seurantaan ja Early Access -julkaisuun, kun taas konsolijulkaisuilla on omat vaatimuksensa ja prosessinsa.

Jälkituotantovaihe alkaa pelin julkaisun jälkeen. Tässä vaiheessa keskitytään pääasiassa pelin ylläpitoon ja kehityksen viimeistelyyn. Pelitestaajista huolimatta useimmissa peleissä on vielä julkaisun jälkeenkin pieniä ohjelmointivirheitä (bugeja), joten ensimmäiset kuukaudet käytetään yleensä virheiden tunnistamiseen ja korjaamiseen (Nuclino, n.d.). Lisäksi kehittäjät voivat julkaista päivityksiä, lisäsisältöä (DLC) tai parannuksia, jotka tasapainottavat pelikokemusta. Pelaajayhteisön palaute on tärkeää pelin kehittämisessä, sillä se auttaa tunnistamaan ongelmakohtia, joita testausvaiheessa ei välttämättä havaittu. Joissakin tapauksissa tiimi voi myös aloittaa seuraavan projektin suunnittelun jo tässä vaiheessa. Pitkän aikavälin tuki on monille peleille elintärkeää.

Päivitykset ja uudet ominaisuudet voivat parantaa pelin elinikää ja myyntiä, sekä ylläpitää aktiivista pelaajakuntaa pitkään julkaisun jälkeen.

7 PELIPROJEKTI – CASTLEXPLOR

7.1 Projektin eteneminen

Projektin toteutuksessa hyödynsin sovellettua versiota vesiputousmallista, missä yhdistin perinteisen mallin vaiheittaisen etenemisen ja ketterän kehityksen mukaisen joustavuuden. Perinteinen vesiputousmalli ei yksinään soveltunut täysin peliprojektin tarpeisiin, koska pelinkehityksessä muutostarpeet ovat yleisiä. Esimerkiksi pelimekaniikat voivat muuttua tai tarkentua testauksen aikana. Tämän vuoksi muokkasin mallia siten, että jaoin tuotantovaiheen sprintteihin. Näissä sprinteissä keskityin yhteen osa-alueeseen kerrallaan (esim. liikkuminen, vihollisten tekoäly, esineiden käyttö) ja sen jälkeen testasin lopputuloksen ennen seuraavaan sprinttiin siirtymistä. Tämän lähestymistavan ansiosta pääsin testaamaan toiminnallisuuksia heti, eivätkä mahdolliset ongelmat kasautuneet loppuvaiheeseen, kuten perinteisessä vesiputousmallissa saattaa tapahtua. Tämä tuki projektin etenemistä ja vähensi ohjelmointiin liittyviä riskejä. Projektin alkuvaiheessa määrittelin seuraavat keskeiset tavoitteet:

- tutoriaalialue ja yksi taso
- pelaajan liikkuminen y- ja x-akseleilla
- vihollisen liikkuminen pelaajaa kohti pelaajan lähestyessä vihollista
- pelaajan ja vihollisen vahingon vastaanottaminen ja taistelu
- osumiseen reagoivat esineet
- pelin tilan tallentaminen tason vaihtuessa

Mekaniikat, jotka lisäisin peliin ajan salliessa:

- teksti-ilmoitukset
- kokemuspisteet pelaajan tuhotessa vihollisen
- aseen päivittäminen
- resurssien kerääminen (kulta)
- päivittyvä xp -palkki ja pelaajan informaationäkymä

Projekti kesti kokonaisuudessaan noin neljä viikkoa, joista ensimmäisen käytin grafiikkojen suunnitteluun ja toteutukseen Aseprite-ohjelmalla. Loput ajasta keskityin pelilogiikan ohjelmointiin Unityllä. Projektin taustalla oli jo aiemmin hahmottelemani peli-idea, joka nopeutti suunnittelua ja auttoi minua keskittymään toteutuksen tärkeimpiin osiin.

Työn jakaminen sprintteihin auttoi minua saamaan tarvittavat grafiikat valmiiksi ennen ohjelmointivaiheeseen siirtymistä. Tämän ansiosta vältyin jatkuvalta hyppimiseltä grafiikan ja ohjelmoinnin välillä, mikä olisi hidastanut työskentelyäni ja vaikuttanut negatiivisesti keskittymiseeni. Tein kaikki grafiikat itse enkä käyttänyt valmiita resursseja. Piirsin esimerkiksi hahmot, seinät ja esineet Aseprite-sovelluksella sekä viimeistelin ne ennen pelissä hyödyntämistä. Tämä mahdollisti sen, että ohjelmoinnin aikana pystyin keskittymään pelilogiikkaan ja toiminnallisuuteen ilman jatkuvaa tarvetta palata takaisin grafiikkatyöhön suunnittelemaan tarvittavia spritejä. Sprite on pelissä käytettävä yksittäinen kuva, joka näkyy ruudulla. Se voi olla esimerkiksi pelaajahahmo, vihollinen tai esine.

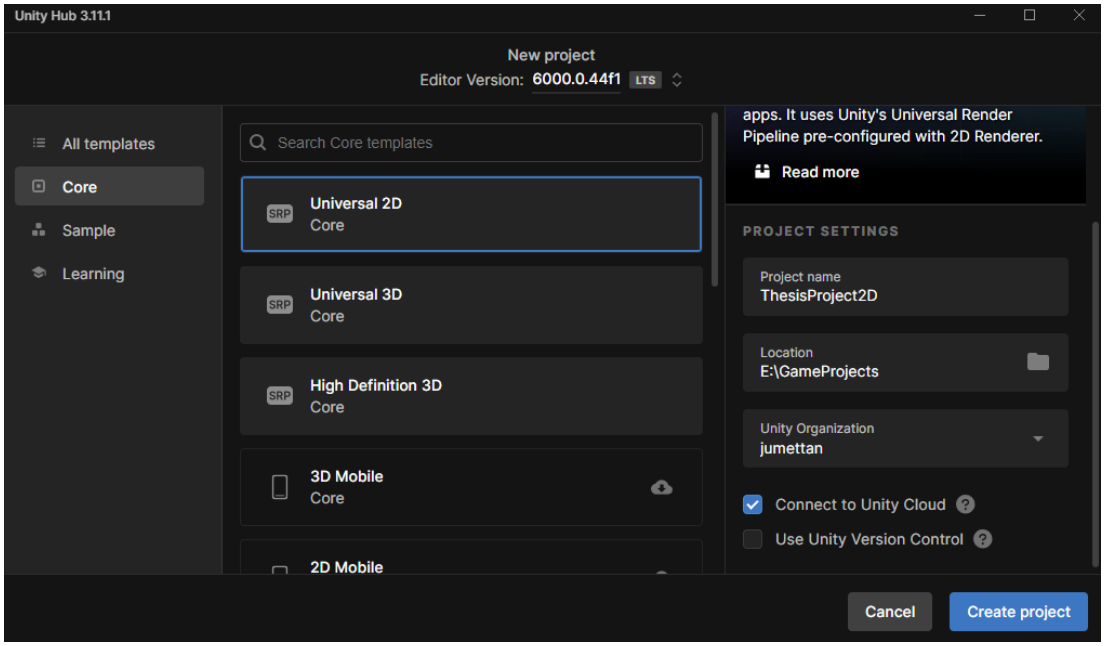
Kehityksen aikana ilmeni kuitenkin muutamia tilanteita, joissa täytyi tehdä pieniä visuaalisia korjauksia tai säätöjä, kuten animaatioiden hienosäätöä tai kentän elementtien sijoittelua. Tällöin palasin Asepriteen tekemään tarvittavat muutokset. Nämä hienosäädöt eivät rikkoneet sprinttirakennetta, vaan osoittivat minulle sen, että edes hyvällä suunnittelulla ei voi ennakoida kaikkea ja kokonaisuuden kannalta käytännön testaus on erittäin olennaista. Valmiiden resurssien ansioista muokkaaminen oli kuitenkin mielestäni nopeaa ja helppoa.

7.2 Projektin aloitus

Unity -projektin hallintaan käytetään Unity Hubia, jonka voi ladata Unityn kotisivuilta. Asennuksen jälkeen sovellus avataan ja siirrytään kohtaan Installs, jossa valitaan Unityn uusin vakaa versio. Uusimman version käyttö on

suositeltavaa aina, kun aloitetaan uusi projekti. Tämän projektin aloitushetkellä uusin versio oli Unity 6. Seuraavaksi valitsin kehitystyäkalun, jolla ohjelmoin pelin toiminnallisuudet. Valitsin Microsoft Visual Studion, joka on yleisesti käytetty ja integroituu hyvin Unityyn sekä toimii myös oletusvalintana. Jos olisin halunnut käyttää esimerkiksi JetBrains Rideria tai Visual Studio Codea sen olisi voinut vaihtaa myöhemmin asetuksista. Tässä vaiheessa olisi voinut myös ladata lisämoduuleita, kuten tuen eri alustoille (esim. mobiilipelien kehitys), kielipaketit ja Unityn dokumentaatio. Koska tämä projekti kehitettiin tietokoneelle, lisämoduuleita ei tarvinnut valita, sillä Unity tukee oletuksena tietokoneella pelattavia 2D-pelejä.

Kun Unity oli asennettu, siirryin New Project -osioon. Valitsin Universal 2D -mallipohjan, joka soveltuu hyvin 2D-pelien kehittämiseen. Projektille täytyi antaa nimi ja Unity Organization kohtaan valita oma Unity -käyttäjä. Lopuksi määritettiin tallennuspaikka ja luotiin projekti (kuva 3). Tämän jälkeen Unity Editor avautuu automaattisesti ja kehitystyö voi alkaa.



Kuva 3. Kuvakaappaus Unity projektin luomisesta.

7.3 Peliprojektin konsepti ja tavoitteet

Ideana oli rakentaa isometrinen 2.5D-toimintaroolipelin demo, jossa pelaaja pyrkii pääsemään useiden vihollisten täyttämien kerrosten läpi linnoituksen huipulle. Projektin tavoitteena oli kehittää toimiva ja pelattava prototyyppi, jossa painotettiin erityisesti pelattavuutta, taistelumekaniikkoja ja ympäristön tutkimista. Demossa halusin tuoda esiin selkeän etenemisrakenteen, jossa jokainen kerros tarjoaa uudenlaista haastetta niin vihollisten kuin kenttäsuunnittelun osalta. Yksi tavoitteistani oli myös testata, kuinka hyvin isometrinen näkymä ja 2.5D-grafiikka yhdistyvät sujuvaksi pelikokemukseksi.

Pelin suunnittelu alkoi heti työn alussa. Minulla oli jo perustason idea, joka oli hiomista vaille valmis. Alussa koin oleellisimmaksi ottaa ideastani ylös tärkeimmät piirteet, joiden toteutus oli listalla ensimmäisenä. Ideat, jotka eivät päässeet listalle, jäivät odottamaan peliin lisäystä mahdollisuuksien mukaan. Tällä tavoin vältyin keskittymästä epäolennaisiin asioihin projektin alussa. Tämä helpotti työn tekemistä ja projektin etenemistä huomattavasti. Pelimekaniikoissa keskityin erityisesti roolipeleissä keskeisessä roolissa olevaan pelaajan etenemismekaniikkaan, jossa hahmo kehittyy ja saa uusia varusteita kerrosten edetessä. Seuraavaksi keskityin taistelu- ja kokemuspistemekaniikkaan, koska pelaaja tarvitsee erilaisia esteitä kehittyäkseen. Tämä taas johti tarpeeseen palkita pelaaja onnistuneesta vihollisen voittamisesta. Valitsin palkinnoksi kullan, jolla pystyy päivittämään aseen ominaisuuksia.

Mielestäni tämä idea oli paras kaikista keksimistäni vaihtoehdoista. Pelimekaniikat luovat kierteen tarpeesta kehittyä ja saavuttaa uusia korkeuksia. Tasojen tutkiminen taas palkitsee pelaajaa rahapalkinnoilla, joita käytetään aseen päivittämiseen. Viholliset tuovat peliin haastetta ja edistävät hahmon kehitystä pelissä. Pelaajan edetessä tasoilla vastaan tulee vaikeampia vihollisia ja alueita, joiden päihittäminen vaatii kärsivällisyyttä. Jokaiset taistelut palkitsevat pelaajaa jollain tavalla oli se sitten kokemuspisteillä tai kullalla. Näin pelaaja voi kohdata ja päihittää yhä vaikeampia haasteita matkallaan kohti linnoituksen huippua.

7.3.1 Projektin aikataulutus

Suunnittelin projektin aikataulutuksen alusta alkaen realistisesti ja ottamalla huomioon käytettävissä olevan ajan sekä oma osaamisen. Työn jaoin eri vaiheisiin viikoittain ja jokaisella oli selkeä tavoite ja tehtäväkokonaisuus.

Taulukko 2. Projektin vaiheet ja aikataulu

Vaihe	Ajankohta	Tehtävät
Esituotanto	Viikko 1	- Pelikonsepti ja peli-idean hiominen - Pelisuunnittelu - Visuaalinen suunnittelu ja tyylin päättäminen
Tuotanto	Viikko 2 ja 3	- 2.5D seinät ja lattia - Pelaajan hahmo ja viholliset - Muut objektit ja resurssit - Tutoriaalialueen suunnittelu Unitylla - Pelaajan ja vihollisen liikkuminen - Ensimmäisen tason suunnittelu - Toiminnalliset esineet - Tallentaminen ja lataaminen muistista - Tekstit (vahinkoilmoitukset, kokemuspisteet ja kulta) - Taistelumekaniikat - Lisäanimaatiot Unity Animaattorilla - Pelaajan informaatiovalikko
Testaus	Viikko 4	- Jokaisen sprintin jälkeen toimivuuden testaus - Hienosäätäminen (aseen vahinkomäärä ja vihollisen nopeus) - Yleisiä korjauksia

Lopuksi viimeistelin projektin pienillä säädöillä, kuten vihollisten vaikeustason tasapainottamisella, virheiden korjauksella ja käyttöliittymän viimeistelyllä. Aikataulua jouduin jonkin verran mukauttamaan projektin aikana, mutta

kokonaisuutena eteneminen pysyi suunnitelman mukaisena. Koska projekti oli yhden henkilön toteuttama, oli aikataulu ja konsepti sekä toteutus tietoisesti yksinkertainen ja helposti hallittava.

7.4 Pelinkehitys aloittaminen graafisella suunnittelulla

Aloitin kehitysprosessin hahmojen ja ympäristön grafiikkojen suunnittelulla. Tavoitteena oli luoda peliin vanhaa ja synkkää menneisyyttä huokuva ilmapiiri, joka tukisi jännittävää ja mysteeristä tunnelmaa. Valitsin grafiikkatyyliksi pikseligrafiikan, koska se sopii hyvin pelin estetiikkaan ja auttoi luomaan halutun visuaalisen ilmeen. Pikseligrafiikka toi peliin myös retrohenkisen tunnelman, joka tukee hyvin teemaa vanhasta ja rappeutuneesta maailmasta.

Suunnittelin pelin hahmot ja ympäristöt yksinkertaisiksi, jotta ne eivät veisi liikaa huomiota pelattavuudelta, mutta samalla säilyttäisivät pelaajan mielenkiintoa herättävän visuaalisen ilmeen. Hahmojen animaatiot ja liikkeet pyrin pitämään sulavina ja selkeinä, jotta peli tuntuisi miellyttävältä pelata. Ympäristön grafiikkojen tarkoitus oli tukea pelaajan kokemusta etenemisestä kohti vaarallista ja tuntematonta aluetta. Suunnittelin ympäristön tulevaisuutta ajatellen niin, että mahdolliset lisätasot erottuisivat visuaalisesti toisistaan omilla erityispiirteillään. Näin pelaaja saisi etenemisen tunteen kohdatessaan uusia ja kiinnostavia alueita.

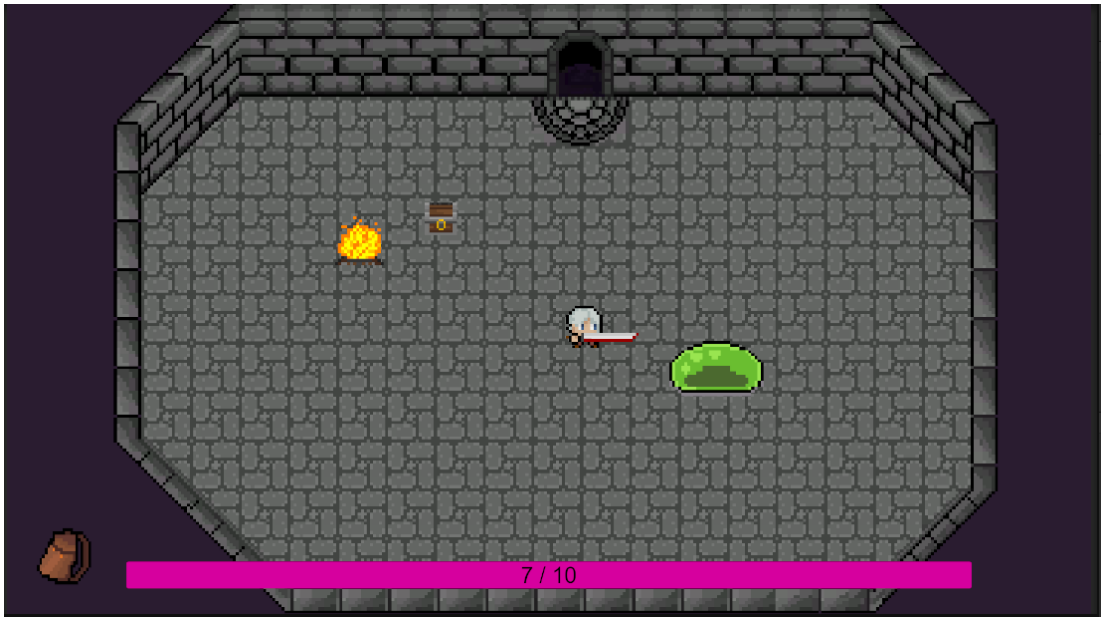
Grafiikoiden luomiseen käytin Aseprite-ohjelmaa. Itse tehdyt grafiikat antoivat täyden kontrollin pelin visuaaliseen ilmeeseen. Graafisen suunnittelun tavoitteena oli varmistaa, että peli on visuaalisesti houkutteleva mutta myös toiminnallisesti selkeä, jotta pelaaja voi keskittyä täysin pelaamiseen ilman, että visuaaliset elementit häiritsevät pelikokemusta. Yksinkertaisilla ja selkeillä grafiikoilla pyrittiin tukemaan pelin teemaa ja tunnelmaa ilman liiallista visuaalista kuormitusta (kuva 4).

Kuva 4. Kuvakaappaus luoduista pelin grafiikoista tasoon 1.

7.4.1 Testialueen rakentaminen

Grafiikoiden suunnittelun ja toteuttamisen jälkeen siirryin Unityn pariin ja aloitin pelin testialueen rakentamisen. Testialueen tarkoituksena oli testata pelimekaniikkojen toimivuutta heti niiden ohjelmoinnin jälkeen sekä luoda visuaalinen hahmotelma siitä, miltä pelaajan perusnäkymä pelissä näyttäisi. Ensimmäisenä asetettiin pelaajan hahmo pelikohtaukseen (Scene), jonka jälkeen luotiin uusi 2D-tilemap pelialueen sekä kenttärakenteiden rajaamista varten. Tämä mahdollisti yksittäisten laattojen sijoittelun ja erilaisten alueiden, kuten lattioiden ja seinien visuaalisen ja toiminnallisen hahmottamisen. Asettelin kaikki resurssit pelikohtaukseen jo valmiiksi, jotta saisin parhaan mahdollisen kuvan mieleeni, miltä pelaajan näkymä tulisi tulevaisuudessa näyttämään (kuva 5). Tein Sceneen uuden Tilemap Grid -objektin, jonka sisälle lisäsin kolme lapsiobjektia. Nimesin uudet objektit niiden käyttötarkoituksen mukaisesti, jotka olivat lattia, seinä ja katto. Seiniin lisäsin Tilemap Collider 2D-komponentin, jotta kaikki sillä valinnalla maalatut kohteet estäisivät pelaajan liikkeen sen läpi. Kulmien Collider laatikot piti muokata niin, että ne kattoivat vain puolet neliöstä jakamalla ne kahtia kulmasta kulmaan. Kattoa hallitsevan objektin asettelin niin, että se on latausjärjestyksessä pelaajan jälkeen. Tällöin pelaajan hahmo voi kulkea siitä läpi ja näyttää siltä, että kulkeminen tapahtuisi sen alta. Lattia

oli kaikista ensimmäisenä latausjärjestyksessä, sillä kaikki peliin tulevat objektit ovat sen päällä.



Kuva 5. Kuvakaappaus pelin testialueesta.

7.4.2 Pelaajan liikkuminen ja törmäyksen tunnistaminen

Ennen varsinaisen liikkumiskoodin ohjelmointia aloitin järjestämään pelin visuaalista näkymää ja objektien piirtojärjestystä. Lisäsin kaksi uutta Sorting Layeria: Blocking -kerroksen, jota käytetään rajoittamaan pelaajan liikkumista ja Actors -kerroksen pelaajan, vihollisten sekä muiden liikkuvien hahmojen spriteille. Tämä jako auttaa erottelemaan paikallaan olevat liikkumisen estävät objektit liikkuvista hahmoista, kuten pelaaja ja vihollinen.

Sorting Layer -toiminnolla varmistetaan, että hahmot ladataan oikeassa järjestyksessä suhteessa taustaan ja muihin objekteihin. Tällä tavoin vältetään tilanne, jossa esimerkiksi pylväs tai muu este piirtyisi hahmon päälle virheellisesti. Jokaiselle sprite -komponentille voidaan erikseen määrittää, mihin Sorting Layeriin se kuuluu ja mikä sen järjestys kyseisessä kerroksessa on (Order in Layer).

Tämän jälkeen valitsin Player objektin ja lisäsin sille uuden MonoBehaviour skripti komponentin inspectorissa, jonka nimeksi annoin Player. Halusin rakentaa pelaajan liikkumisen ja esteen tunnistuksen siten, että sitä pystyisi myöhemmin käyttämään myös vihollisessa pienten muokkausten jälkeen. Tämän rakenteen avulla pelaajan liikkumista voidaan hallita selkeästi ja luotettavasti. Liikkumista rajoittavat esteet tunnistetaan BoxCastin avulla, jolloin pelaaja ei pääse kävelemään läpi seinistä tai muista objektien kolmiulotteisista muodoista, jotka on määritelty Blocking- tai Actor -kerroksille. Lisäksi hahmon kääntyminen vasemmalle ja oikealle hoidetaan peilaamalla sen mittakaavaa paikallisesti, mikä helpottaa pelaajan animaatioiden hallintaa ilman tarvetta erilliselle logiikalle suunnan mukaan.

Koska liikkumistoiminto on rakennettu omaan TryMove-metodiin, tätä logiikkaa voi helposti käyttää myös muissa olioissa. Tämä mahdollistaa sen, että esimerkiksi vihollinen voi käyttää samaa liikkumistapaa kuin pelaaja. Myöhemmin tarkoitukseni on yhdistää tämä kuvassa 6 oleva liikkumislogiikka MovingObject -pohjaluokkaan, jolloin siitä tulisi yhteinen alusta sekä pelaajalle että vihollisille. Näin voin luoda helposti uusia liikkuvia ja esteitä tunnistavia olentoja ilman koodin toistoa, mikä helpottaa pelin jatkokehitystä ja laajentamista merkittävästi.

```
public class Player : MonoBehaviour
{
    private BoxCollider2D boxCollider;
    private Vector3 deltaMove;
    private RaycastHit2D hit;

     private void Awake()
    {
        boxCollider = GetComponent<BoxCollider2D>();
    }
     private void TryMove(Vector2 direction, Vector3 translation)
     {
         hit = Physics2D.BoxCast(transform.position,
         boxCollider.size, 0, direction.normalized
         Mathf.Abs(direction.magnitude * Time.deltaTime),
         LayerMask.GetMask("Actor", "Blocking"));
         if (hit.collider == null)
         {
             transform.Translate(translation * Time.deltaTime);
         }
     }
     private void FixedUpdate()
     {
         float x = Input.GetAxis("Horizontal");
         float y = Input.GetAxis("Vertical");
         deltaMove = new Vector3(x, y, 0);

         if(deltaMove.x < 0){
             transform.localScale = Vector3.one;
         }else if(deltaMove.x > 0){
             transform.localScale = new Vector3(-1, 1, 1);
         }
         TryMove(Vector2.up * deltaMove.y, Vector3.up * deltaMove.y);
         TryMove(Vector2.right * deltaMove.x, Vec-tor3.right * deltaMove.x);
     }
}
```

Kuva 6. Kuvakaappaus Player-luokasta.

Awake() on Unityn sisäänrakennettu metodi, joka suoritetaan ensimmäisenä, kun skripti aktivoituu. Metodin kutsuminen tapahtuu vain kerran pelin alussa ja se suoritetaan ennen kaikkia muita päivitysmetodeja. Player-luokassa sen tarkoituksena on hakea hahmon BoxCollider2D ja tallentaa se muuttujaan. Tällä varmistetaan, että BoxCollider muuttuja on määritelty ennen muun skriptin suorittamista. FixedUpdate() puolestaan suoritetaan säännöllisin väliajoin fysiikkamoottorin päivitysten tahdissa. Sen tarkoituksena ei ole päivittää peliä yleisesti, vaan hoitaa fysiikkaan liittyviä toimintoja kuten liikkeen, törmäysten ja voimien käsittelyä. Esimerkiksi hahmon tai vihollisen liike toteutetaan usein FixedUpdate()-metodin avulla, jotta se toimisi sulavasti ja tarkasti fysiikkalaskennan kanssa.

7.4.3 Toiminnalliset esineet

Toiminnallisissa esineissä käytin hyödykseni perintää ohjelmoinnin toistamisen sijaan. Loin yleisen pohjarakenteen, jota muut esineet voivat hyödyntää. Tällä tavoin esimerkiksi kerättävät esineet, ovet ja portaalin kaltaiset objektit voivat kaikki toimia yhteisen koodin pohjalta, mikä taas säästi kallisarvoista aikaa.

Collide-luokka toimi pohjana kaikille törmäyksiä käsitteleville esineille. Sen ytimessä on Overlap()-metodi, joka tarkista, osuuko jokin määritellyistä kohteista (määritetty ContactFilter2D-filtterillä) tämän objektin päälle. Jos törmäys havaitaan, kutsutaan OnCollision()-metodia. Tämä metodi on tarkoitettu ylikirjoitettavaksi perivissä luokissa, jolloin jokainen objekti voi määritellä oman toimintansa törmäyksessä. Esimerkiksi arkku avautuu ja antaa kultaa siihen osuessa. Tämä rakenne mahdollisti koodin uudelleenkäytön, helpotti toiminnallisten objektien lisäämistä peliin ja teki samalla järjestelmästä laajennettavan tulevaisuudessa. Start() on taas Unityn sisäänrakennettu metodi, joka suoritetaan ensimmäisen kerran pelissä, kun objekti aktivoituu (kuva 7). Se suoritetaan ennen Update()-metodia, joka vastaa pelin jatkuvasta toiminnasta. Start() hakee objektin BoxCollider2D-komponentin ja tallentaa sen muuttujaan. Tämä varmistaa, että Update()-metodi voi käyttää kyseistä komponenttia ilman virheitä, koska se riippuu boxCollider-muuttujasta toimiakseen oikein.

```
public class Collide : MonoBehaviour
{
    public ContactFilter2D filter;
    private BoxCollider2D boxCollider;
    private Collider2D[] hits = new Collider2D[10];
    protected virtual void Start()
    {
        boxCollider = GetComponent<BoxCollider2D>();
    }
    protected virtual void Update()
    {
        boxCollider.Overlap(filter, hits);
        ProcessCollisions();
    }
     private void ProcessCollisions()
    {
        for (int i = 0; i < hits.Length; i++)
        {
            if (hits[i] != null)
            {
                OnCollision(hits[i]);

                hits[i] = null;
            }
        }
    }

    protected virtual void OnCollision(Collider2D hit)
    {
        Debug.Log("Collision not implemented");
    }
}
```

Kuva 7. Kuvakaappaus Collide-luokasta.

Tein myös tasojen vaihtamiseen tarkoitetun objektin, joka vastasi siirtymisen ohella pelitilan tallentamisesta. Tein objektin läpinäkyväksi ja rajasin sen lisäämällä sille BoxCollider2D -komponentin törmäyksen tunnistusta varten. Tämäkin luokka perii Collide-luokan ja määrittelin sille oman toiminnan törmäyksen yhteydessä. Määrittelin sille myös oman käyttäytymisen törmäyksissä ylikirjoittamalla (override) OnCollision-metodin.

Collectable-luokka laajentaa Collide-luokkaa ja määrittelee toiminnan esineille, jotka ovat kerättävissä (kuva 8). Kun pelaaja osuu kerättävään esineeseen kuten arkku, OnCollision()-metodi kutsuu onCollect()-metodia, joka merkitsee esineen kerätyksi asettamalla Collected-muuttujan arvoksi true. Muissa luokissa voidaan laajentaa tätä luokkaa määrittämällä oma keräystoiminto.

```
public class Collectable : Collide
{
    protected bool Collected;

    protected override void OnCollision(Collider2D coll)
    {
        if (coll.name == "Player")
            onCollect();
    }
    protected virtual void onCollect()
    {
        Collected = true;
    }
}
```

Kuva 8. Kuvakaappaus Collectable-luokasta, joka perii Collide-luokan ominaisuudet.

Chest-luokka laajentaa Collectable-luokkaa ja määrittelee toiminnon, jossa pelaaja voi kerätä kultaa arkusta (kuva 9). Kun arkku kerätään, sen sprite päivitetään tyhjäksi ja pelaajan kultamäärä kasvaa määritellyn GoldAmount-arvon mukaan ja tallentaa sen GameManageriin.

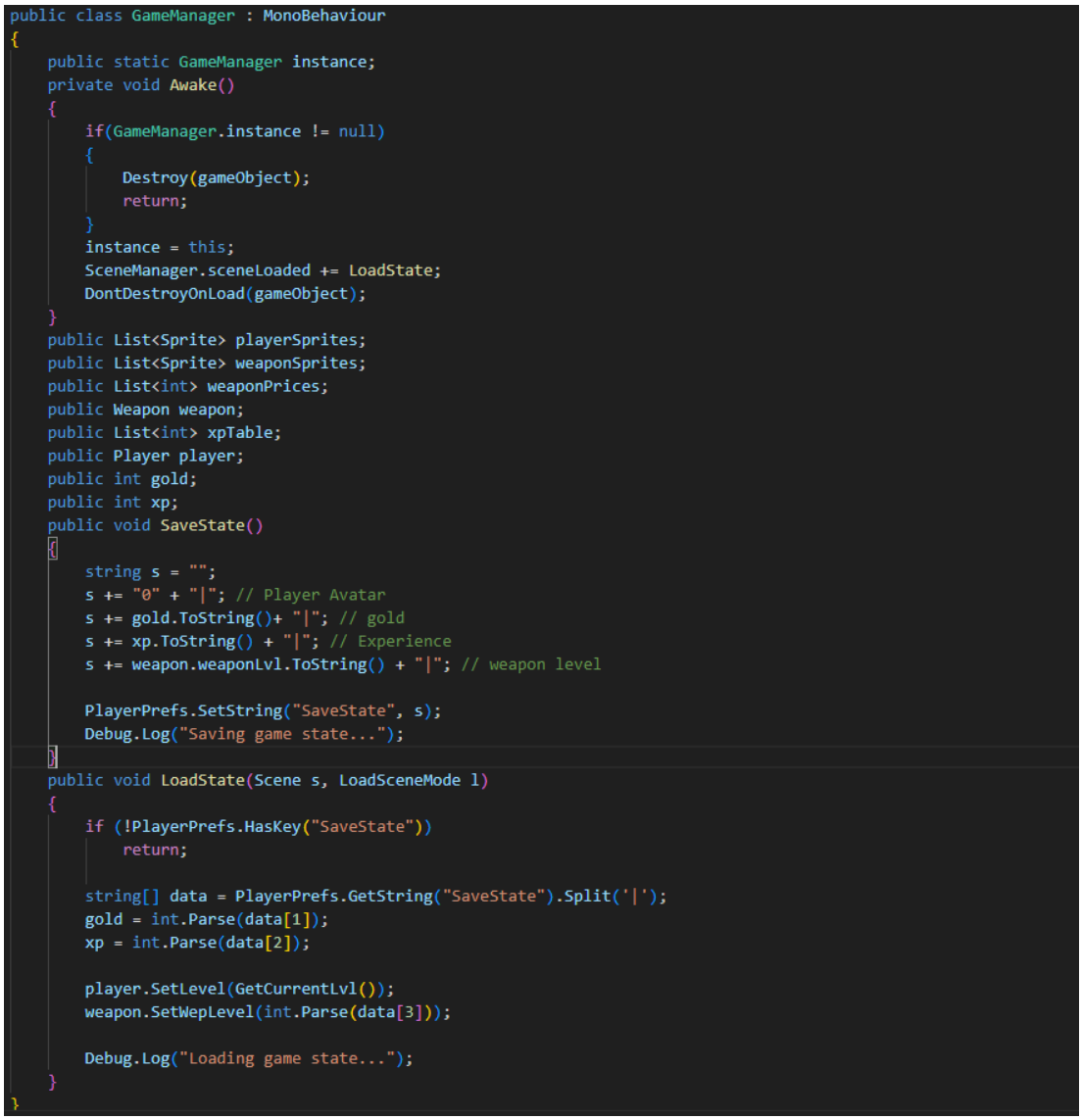
```
public class Chest : Collectable
{
    public Sprite emptyChestSprite;
    public int GoldAmount;
    protected override void onCollect()
    {

        if(!Collected){
            Collected = true;
            GetComponent<SpriteRenderer>().sprite = emptyChestSprite;
            GameManager.instance.gold += GoldAmount;
        }
    }
}
```

Kuva 9. Kuvakaappaus Chest-luokasta, joka perii Collectablen.

7.4.4 GameManager

Loin peliin uuden objektin nimeltä GameManager, johon yhdistin GameManager skriptin. GameManager-luokka toimi pelin keskeisenä hallintaluokkana, joka vastaa tärkeistä julkisista muuttujista ja toiminnoista, kuten pelaajan kultamäärästä, kokemuspisteistä, aseen tasosta sekä pelitilan tallennuksesta ja latauksesta. Se mahdollistaa tiedon säilyttämisen kenttien välillä ja varmistaa, että pelitilanne jatkuu johdonmukaisesti latauksen jälkeen.

Tämä koodi muodostaa keskeisen osan pelin hallintajärjestelmää GameManager-luokan kautta (kuva 10). Se varmistaa, että pelissä on vain yksi GameManager-instanssi käytössä kerrallaan. Awake()-metodissa tarkistetaan, onko instanssi jo olemassa. Jos on, todetaan uusi GameManager-olio turhaksi ja se tuhotaan. Jos ei ole, nykyisestä oliosta tehdään pysyvä instanssi käyttämällä DontDestroyOnLoad()-metodia, joka estää olion tuhoutumisen kenttien välillä.



```
public class GameManager : MonoBehaviour
{
    public static GameManager instance;
    private void Awake()
    {
        if(GameManager.instance != null)
        {
            Destroy(gameObject);
            return;
        }
        instance = this;
        SceneManager.sceneLoaded += LoadState;
        DontDestroyOnLoad(gameObject);
    }
    public List<Sprite> playerSprites;
    public List<Sprite> weaponSprites;
    public List<int> weaponPrices;
    public Weapon weapon;
    public List<int> xpTable;
    public Player player;
    public int gold;
    public int xp;
    public void SaveState()
    {
        string s = "";
        s += "0" + "|"; // Player Avatar
        s += gold.ToString()+ "|"; // gold
        s += xp.ToString() + "|"; // Experience
        s += weapon.weaponLvl.ToString() + "|"; // weapon level

        PlayerPrefs.SetString("SaveState", s);
        Debug.Log("Saving game state...");
    }
    public void LoadState(Scene s, LoadSceneMode l)
    {
        if (!PlayerPrefs.HasKey("SaveState"))
            return;

        string[] data = PlayerPrefs.GetString("SaveState").Split('|');
        gold = int.Parse(data[1]);
        xp = int.Parse(data[2]);

        player.SetLevel(GetCurrentLvl());
        weapon.SetWepLevel(int.Parse(data[3]));

        Debug.Log("Loading game state...");
    }
}
```

Kuva 10. Kuvakaappaus GameManager-luokasta, joka tallentaa ja lataa pelin tilan.

Lisäksi SceneManager.sceneLoaded += LoadState; rivi rekisteröi pelitilan latausfunktion LoadState, joka aktivoituu aina uuden kentän latauksen yhteydessä. Tämän ansiosta pelaajan tila, kuten kultamäärä ja kokemus, säilyy kenttien välillä. Julkiset kenttämuuttujat pitävät sisällään pelissä tarvittavia resursseja ja tilatietoa, kuten hahmojen ja aseiden spritet, aseiden hinnat, aseviittauksen (weapon), kokemustaulukon (xpTable), pelaajaviittauksen (player), sekä kultaa ja kokemuspisteitä kuvaavat muuttujat. Näitä käytetään muun muassa pelaajan kehitykseen, visuaaliseen esittämiseen ja pelin etenemisen seurantaan.

SaveState()-metodi tallentaa pelin nykyisen tilan Unityn PlayerPrefs-järjestelmään merkkijonona, jotta se voidaan myöhemmin ladata uudelleen esimerkiksi pelin käynnistyksen tai kentän latauksen yhteydessä. Metodin alussa luodaan tyhjä merkkijono s, johon kootaan pelin tilaa kuvaavat arvot. Ensimmäiseksi lisätään kovakoodattu arvo "0", joka on varattu pelaajan valitsemalle hahmon ulkonäölle, vaikka sitä ominaisuutta en ehtinyt peliin lisäämään. Tämän jälkeen merkkijonoon lisätään pelaajan nykyinen kultamäärä, kokemuspisteet sekä käytössä olevan aseen taso. Jokaisen arvon perään lisätään erotinmerkki |, jotta arvot voidaan myöhemmin erotella ja lukea ladatessa. Kun kaikki tarvittavat tiedot on koottu, merkkijono tallennetaan PlayerPrefs-muistiin avaimella "SaveState". Lopuksi konsoliin tulostetaan debug-viesti, joka vahvistaa pelitilan tallennuksen onnistuneen. Tämä yksinkertainen rakenne on helposti laajennettavissa, jos pelitilaan halutaan tulevaisuudessa tallentaa lisää tietoa.

LoadState()-metodi vastaa tallennetun pelitilan palauttamisesta, kun peli tai kenttä käynnistyy. Se ottaa parametreina Unityn Scene- ja LoadSceneMode-oliot ja on tarkoitettu liitettäväksi Unityn kentänlataustapahtumaan. Metodin alussa tarkistetaan onko PlayerPrefs-muistiin tallennettu arvo nimeltä "SaveState". Jos tallennetta ei ole, metodi keskeytetään heti. Jos tallennettu tila kuitenkin löytyy, se haetaan merkkijonona ja jaetaan osiin Split('|')-metodilla. Jokainen osio sisältää pelin tilan kannalta keskeisiä tietoja, kuten kullan määrä (data[1]), kokemuspisteet (data[2]) ja aseen taso (data[3]). Nämä muunnetaan takaisin kokonaisluvuiksi ja asetetaan pelihahmon arvoiksi. Pelaajan taso päivitetään metodilla SetLevel(), joka perustuu tämänhetkisiin kokemuspisteisiin ja pelaajan ase päivittyy oikealle tasolle SetWepLevel()-metodin kautta. Lopuksi tulostetaan viesti konsoliin, joka vahvistaa, että pelitila on ladattu onnistuneesti.

7.4.5 MovingObject-luokka ja vihollinen

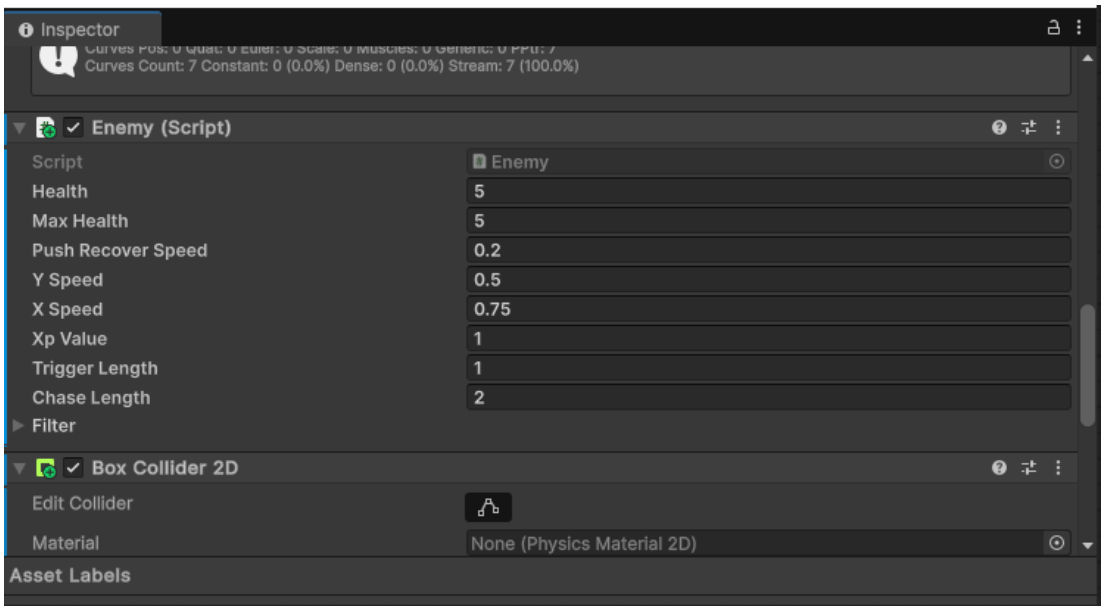
Kuten aiemmin mainitsin, vihollinen tulisi käyttämään samaa pohjaluokkaa kuin pelaaja. Tätä varten tein uuden luokan MovingObject ja kopioin sinne Player-luokan toiminnallisuudet. Player-luokassa hoidettiin tästä eteenpäin

animaatioiden vaihtaminen liikkeen perusteella ja näppäimistöltä tulevat input komennot liikesuunnan määrittämiseksi. Käyttämällä perintää pelaaja perii myös MonoBehaviour-luokan MovingObject kautta.

```
public class Player : MovingObject
{
    private Animator animator;
    private bool isMoving;
    protected override void Start()
    {
        base.Start();
        animator = GetComponent<Animator>();
    }
        private bool IsMoving(float x, float y)
    {
        return x != 0 || y != 0;
    }
    private void FixedUpdate()
    {
        float x = Input.GetAxis("Horizontal");
        float y = Input.GetAxis("Vertical");
        isMoving = IsMoving(x, y);
        if (animator != null)
        {
            animator.SetBool("isRunning", isMoving);
        }
        UpdateMotor(new Vector3(x, y, 0));
    }
}
```

Kuva 11. Kuvakaappaus uudelleen rakennetusta Player-luokasta.

Testasin vielä toimivuuden enne kuin siirryin rakentamaan vihollisen logiikkaa. Tein uuden Enemy skriptin, jonka lisäsin vihollisen komponentteihin. Loin vielä MovingObject luokkaan float muuttujat ySpeed ja xSpeed, jotta vihollisella olisi vakionopeus liikkuessaan. Jälkeenpäin arvoja voi muuttaa pelin sisällä vihollisen inspectorissa (kuva 12).



Kuva 12. Kuvankaappaus valitun vihollisen Inspector-näkymästä.

Vihollisen liikkuminen perustuu ideaan, jossa pelaajan lähestyessä vihollista tämä aktivoituu ja alkaa jahdata pelaajaa. Mikäli pelaaja pääsee tarpeeksi kauas, vihollinen palaa takaisin alkuperäiselle paikalleen. Enemy-luokka perii MovingObject-luokan, jonka kautta se saa käyttöönsä liikkumiseen liittyviä toimintoja, kuten UpdateMotor-metodin. Luokan muuttujia, kuten etäisyysraja ja liikenopeus voidaan säätää Unity-editorissa eri vihollisten tarpeisiin, mikä tukee uudelleenkäytettävyyttä.

Start-metodissa alustetaan vihollisen lähtöpaikka, viittaus pelaajaan sekä haetaan törmäyslaatikko (BoxCollider2D) lapsiobjektista. Kuvassa 13 tarkastellaan FixedUpdate-metodilla pelaajan etäisyyttä. Kun pelaaja on triggerLength-rajan sisällä, vihollinen aloittaa jahtaamisen. Jos pelaaja loittonee yli chaseLength-etäisyyden, vihollinen keskeyttää jahtaamisen ja palaa takaisin lähtöpaikkaansa. Liike toteutetaan UpdateMotor-metodilla.

```
protected override void Start()
{
    base.Start();
    playerTransform = GameManager.instance.player.transform;
    startingPosition = transform.position;
    hitbox = transform.GetChild(0).GetComponent<BoxCollider2D>();
}
private void FixedUpdate()
{
    float distanceToPlayer = Vector3.Distance(playerTransform.position, startingPosition);
    if (distanceToPlayer < chaseLength)
    {
        if (distanceToPlayer < triggerLength)
        {
            chasing = true;
        }
        if (chasing)
        {
            if (!collidingWithPlayer)
            {
                Vector3 directionToPlayer = (playerTransform.position - transform.position).normalized;
                UpdateMotor(directionToPlayer);
            }
        }
        else
        {
            Vector3 returnDirection = startingPosition - transform.position;
            UpdateMotor(returnDirection);
        }
    }
    else
        {
            Vector3 returnDirection = startingPosition - transform.position;
            UpdateMotor(returnDirection);
            chasing = false;
        }
```

Kuva 13. Kuvakaappaus Enemy-luokan Start()- ja FixedUpdate()-metodeista.

Vihollinen tarkistaa jatkuvasti törmäyksiä pelaajaan OverlapCollider-metodilla. Törmäys pelaajaan havaitaan tarkistamalla objektille määritellyt leimat. Jos objektilla on molemmat Fighter ja Player_0 leimat, asetetaan collidingWithPlayer-muuttuja todeksi, mahdollistaen tilanteeseen sopivat toiminnot kuten hyökkäys tai pysähtyminen. Lopuksi törmäystaulukko nollataan jokaisen päivityksen jälkeen, jotta seuraava tarkistus voidaan tehdä ilman häiriöitä.

7.4.6 Taistelumekaniikat

Lähtiessäni toteuttamaan taistelumekaniikkoja halusin logiikan toimivan niin, että jokainen liikkuva objekti voi myös taistella. Tämän vuoksi loin uuden luokan nimeltä Fighter, joka toimii pohjana kaikille taistelukykyisille oliolle, kuten pelaajalle ja vihollisille. Fighter-luokka kuvassa 14 sisältää kaikki perustoiminnot, joita tarvitaan vahingon vastaanottamiseen sekä elinvoiman (HP) hallintaan. Jokaisella oliolla on health- ja maxHealth-muuttujat, joiden avulla seurataan sen elinvoimaa. Kun olio ottaa vahinkoa, kutsutaan ReceiveDamage-

metodia, joka vastaanottaa Damage-rakenteen. Tämä sisältää tiedot vahingon määrästä, sen suunnasta ja siihen liittyvästä työntövoimasta (push force).

Luokkaan on toteutettu myös haavoittumattomuusmekaniikka (immuneTime), joka estää esimerkiksi sen, että olio ei voi ottaa vahinkoa vastaan useita kertoja nopeasti peräkkäin ja menettää useita elinvoimapisteitä yhdellä kertaa. Lisäksi puskemismekaniikka mahdollistaa sen, että vahingon ohella olio myös siirtyy osuman suunnan mukaisesti taaksepäin. Tätä varten pushDirection-muuttuja tallentaa suunnan ja voiman, jolla olio työnnetään.

Mikäli olion elämä laskee nollaan tai sen alle, kutsutaan Death()-metodia. Tämä metodi on tarkoituksella virtuaalinen, eli se voidaan korvata kirjoittamalla se jokaisessa oliossa uudelleen. Kuoleman jälkeiset tapahtumat voidaan määritellä erikseen pelaajalle ja vihollisille ilman koodin toistoa. Koska taistelutoiminnallisuus on olennainen osa liikkuvia objekteja, vaihdoin MovingObject-luokan perimään suoraan Fighter-luokan MonoBehaviourin sijaan. Näin kaikki liikkuvat oliot voivat automaattisesti osallistua taisteluun, mikä vähentää koodin monimutkaisuutta ja parantaa uudelleenkäytettävyyttä.

```
public class Fighter : MonoBehaviour
{
    public int health = 10;
    public int maxHealth = 10;
    public float pushRecoverSpeed = 0.2f;
    protected float immuneTime = 1.0f;
    protected float LastimmuneTime;
    protected Vector3 pushDirection;
    protected virtual void ReceiveDamage(Damage damage)
    {
        if(Time.time - LastimmuneTime > immuneTime)
        {
            LastimmuneTime = Time.time;
            health -= damage.damageAmount;
            pushDirection = (transform.position - damage.origin).normalized * damage.pushForce;


            GameManager.instance.ShowText(damage.damageAmount.ToString(), 25, Color.red, transform.position, Vector3.zero, 0.5f);

            if (health <= 0)
            {
                health = 0;
                Death();
            }
        }
    }
    protected virtual void Death()
    {

    }
}
```

Kuva 14. Kuvakaappaus Fighter-luokasta.

Viholliselle on tehty erillinen hitbox-alue, joka vastaa vahingon tuottamisesta pelaajaan (kuva 15). Tämä rakenne toimii erityisen hyvin tilanteissa, joissa pelissä on useita erilaisia vihollistyyppejä, joilla on omat käyttäytymismallinsa, mutta jotka silti hyödyntävät samaa vahingontuottomekaniikkaa.

Koska EnemyHitbox on erillinen komponentti, se voidaan kiinnittää eri kokoisiin vihollisiin ja muokata sen osumaa vastaamaan kunkin vihollisen fyysistä muotoa tai hyökkäyksen aluetta. Tämä mahdollistaa esimerkiksi sen, että isomman vihollisen osuma-alue on laajempi tai että tietyt viholliset tekevät vahinkoa vain silloin, kun ne suorittavat hyökkäysanimaation.

```
public class EnemyHitbox : Collide
{
    public int damage;
    public float pushForce;
    protected override void OnCollision(Collider2D coll)
    {
        if(coll.name == "Player")
        {
            //new damage object -> send to player
            Damage dmg = new()
            {
                damageAmount = damage,
                pushForce = pushForce,
                origin = transform.position
            };
            coll.SendMessage("ReceiveDamage", dmg);
        }
    }
}
```

Kuva 15. Kuvakaappaus EnemyHitbox-luokasta.

Pelaajalle loin hyökkäysanimaation, joka sitten tuottaa puolestaan viholliseen vahinkoa. Pohjana Weapon-luokalle oli enemyHitbox ja pystyin käyttämään samaa logiikkaa myös aseessa. Luokissa on kuitenkin muutama pieni ero, koska pelaajalle täytyi tehdä animaatio, joka aktivoidaan painamalla välilyöntiä. Lisäsin pohjaan vielä ehdon logiikkaan, jotta vältyttäisiin aseen osumista pelaajaan. Laitoin logiikan ympärille vielä uuden if-loopin, joka tarkistaa onko objektin leima Fighter. Sisällä oleva if-loop tarkistaa onko kyseessä pelaaja.

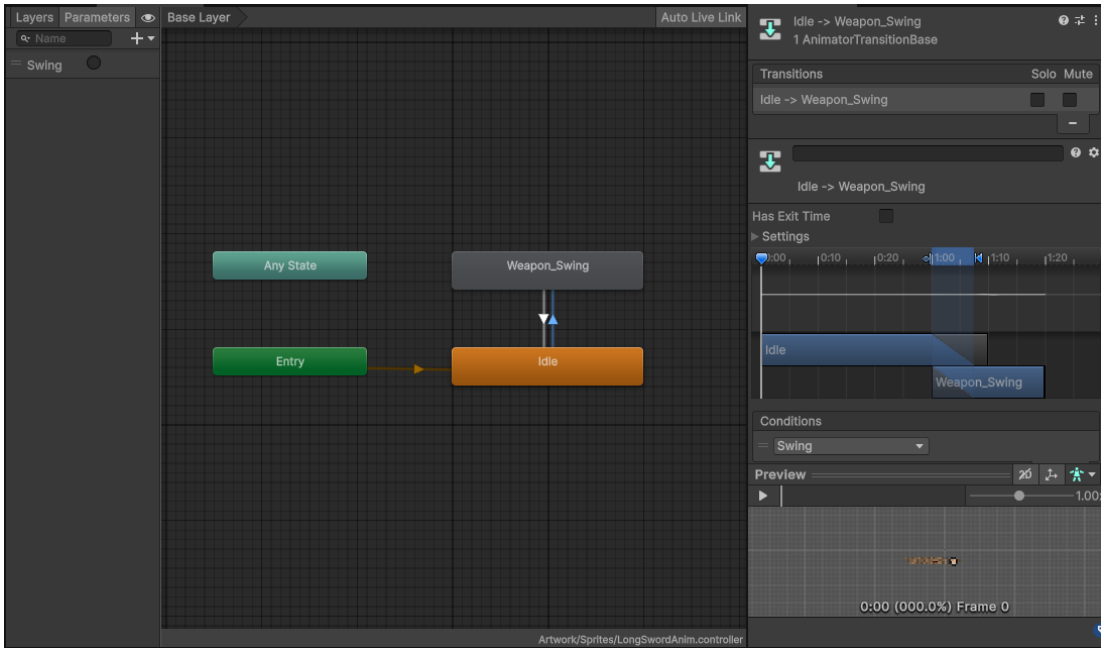
Jos on, niin palataan takaisin käyttämällä returnia. Kun logiikka oli kunnossa, lisäsin toiminnallisuuden hyökkäysanimaation käynnistämiseksi. Unityssa liitin aseen spriten pelaajan hahmon lapsiobjektiksi ja siihen Weapon-skriptin komponentiksi. Aseen käyttö sidottiin välilyöntiin, joka jäi lopulliseksi hyökkäyspainikkeeksi.

Skriptissä haetaan SpriteRenderer ja Animator, jotta ase voidaan tarvittaessa näyttää ja animaatio käynnistää. Update-metodissa tarkistetaan, onko pelaaja painanut välilyöntiä ja onko hyökkäysten välillä kulunut riittävä aika (cool-down). Jos ehdot täyttyvät, käynnistetään Swing()-metodi, joka laukaisee animaation SetTrigger("Swing") -kutsulla (kuva 16). Tämä mahdollistaa tarkasti ajoitetut hyökkäysanimaatiot ja niiden vaikutukset pelissä.

```
private void Awake()
{
    spriteRenderer = GetComponent<SpriteRenderer>();
}
protected override void Start()
{
    base.Start();
    animator = GetComponent<Animator>();

}
protected override void Update()
{
    base.Update();
    if (Input.GetKeyDown(KeyCode.Space))
    {
        if(Time.time - lastSwingTime > cd)
        {
            lastSwingTime = Time.time;
            Swing();
        }
    }
}
private void Swing()
{
    animator.SetTrigger("Swing");
}
```

Kuva 16. Kuvakaappaus lyöntianimaation määrittämisestä.

Tämän jälkeen siirryin toteuttamaan hyökkäysanimaation Unityn omalla Animator-työkalulla. Loogisena jatkona aiemmalle koodille, loin uuden animaatiotilan nimeltä Swing ja määritin sille trigger-parametrin, jonka avulla ase aktivoituu oikeaan aikaan. Animaation siirtymät määriteltiin niin, että Swing käynnistyy heti kun trigger aktivoidaan ja palaa takaisin idle-tilaan animaation päätyttyä. Näin varmistetaan, että hyökkäys tapahtuu saumattomasti osana pelin rytmiä ja visuaalista ilmettä.



Kuva 17. Kuvakaappaus animaatiologiikasta.

7.5 Pelin testaaminen

Testaaminen oli olennainen osa koko projektin kehitystyötä ja sitä tehtiin jatkuvasti jokaisen uuden toiminnallisuuden lisäämisen jälkeen. Tavoitteena oli havaita ja korjata virheet mahdollisimman varhaisessa vaiheessa, jotta suuremmilta ongelmilta loppuvaiheessa vältyttäisiin. Käytännössä peliä testattiin toistuvasti pelin sisäisten tilanteiden kuten liikkumisen, taistelun ja siirtymien osalta. Tästä huolimatta testauksessa ilmeni muutamia ongelmia, jotka vaikuttivat pelattavuuteen. Esimerkiksi taistelutilanteessa havaitsin, että jos vihollinen osui pelaajaan aivan seinän vieressä, pelaaja saattoi työntyä ulos kentän rajatun alueen ulkopuolelle eikä päässyt enää takaisin peliin tarkoitetulle

alueelle. Tämä heikensi pelikokemusta ja vaati korjausta törmäyksentunnistukseen ja rajojen käsittelyyn.

Toinen merkittävä ongelma ilmeni scenen vaihdossa. Olin unohtanut kopioida testikäytössä olleen scenen elementit varsinaiseen peliin siirtymäkohdassa, mikä johti siihen, että uuden alueen viholliset eivät enää toimineet kuten pitäisi eli toisin sanoen niiden logiikka ei aktivoitunut. Tämän korjasin siirtämällä kaikki tarvittavat komponentit ja varmistamalla, että jokaisessa scenessä on oikeat viittaukset ja rakenteet toimivan logiikan takaamiseksi. Nämä kaksi ongelmaa olivat keskeisimmät haasteet testauksen aikana ja niiden ratkaiseminen auttoi varmistamaan pelin vakaamman ja toimivamman version.

Näiden korjaamisen jälkeen peli oli mielestäni pelattavassa kunnossa. Vastaan kuitenkin tuli vielä ohjelmointivirheitä testatessa varsinaista ensimmäistä tasoa. Mutta kaiken kaikkiaan demosta saa hyvän käsityksen miltä peli tulisi tulevaisuudessa näyttämään. Korjailin vielä näitä ongelma sitä mukaan, kun niitä tuli vastaan. Tässä kohtaa huomasin erillisten testaajien vahvuuden pelialalla. Omaa peliä on suhteellisen vaikea testata, koska on tietoinen, miten peliä tulisi pelata. Tämän takia ei välttämättä huomaa kaikkia pieniä virheitä tai ongelmia, joita pelistä tulee varmasti vielä löytymään.

8 POHDINTA

Opinnäytetyön tarkoituksena oli käydä läpi pelikehityksen eri vaiheet yksittäisen pelikehittäjän näkökulmasta. Projektin aikana perehdyin niin suunnitteluun, ohjelmointiin, grafiikkaan kuin testaukseenkin, mikä antoi kokonaisvaltaisen kuvan siitä, mitä pelin rakentaminen tyhjästä käytännössä vaatii. Yksin työskentely tarkoitti sitä, että jokaisesta osa-alueesta täytyi ottaa vastuu itse, mikä oli ajoittain haastavaa mutta samalla erittäin opettavaista. Eri osa-alueiden yhteensovittaminen korosti hyvän rakenteen ja modulaarisen koodin merkitystä, sillä pienetkin muutokset yhdessä järjestelmässä saattoivat vaikuttaa muualle peliin.

Suurimmat haasteet liittyivät pelimekaniikkojen yksityiskohtien toteutukseen ja ohjelmointivirheiden paikantamiseen. Erityisesti törmäyksentunnistukseen, animaatioiden aikataulutukseen ja vihollislogiikkaan liittyvät ongelmat vaativat paljon testaamista ja hienosäätöä. Samalla nämä haasteet myös kehittivät ongelmanratkaisukykyä ja ymmärrystä pelimoottorin toiminnasta. Projektin myötä opin paljon Unityn käytöstä, C#-ohjelmoinnista sekä pelisuunnittelun käytännöistä. Lisäksi sain kokemusta siitä, miten tärkeää on iteratiivinen kehitysprosessi, jatkuva testaus ja dokumentointi. Vaikka lopullinen peli ei ole täysin valmis tai täydellinen, se toimii toimivana prototyyppinä ja osoituksena siitä, kuinka paljon on mahdollista saada aikaan rajallisilla resursseilla ja ajalla.

Suurimpana ongelmana työssä koin sen lopullisen suuruuden, jota en osannut alussa kunnolla arvioida. Toiminnallisessa osuudessa oli vaikeaa päättä, mitkä tehdyistä asioista tulisi mainita tekstissä. Vaikka peli ei ollutkaan kovin laaja, niin toiminnallisuuksia kyllä riitti. Oli vaikea karsia pois pätkiä, jotka eivät olleet oleellisia pelin kannalta ja samalla taas miettiä mitkä olivat. Tulin esimerkiksi tulokseen, jossa teksti-ilmoitukset vahingon saamisesta tulevat näytölle näkyviin hetkeksi ja pelaajan aseen päivitys mekaniikka jäivät mainitsematta. Eli suurin osa mekaniikoista, jotka liittyivät jollain tasolla pelaajan hahmon kehitykseen.

Kokonaisuudessaan projekti vahvisti kiinnostustani pelikehitystä kohtaan ja antoi varmuutta omista taidoistani. Työ osoitti myös sen, miten tärkeää on kärsivällisyys, suunnitelmallisuus ja kyky ratkaista ongelmia itsenäisesti. Tämä kokemus luo hyvän pohjan jatkaa pelikehityksen parissa tulevaisuudessa joko omana projektina tai osana tiimiä.

LÄHTEET

Atlassian. (n.d.). Waterfall Methodology: A Comprehensive Guide. https://www.atlassian.com/agile/project-management/waterfall-methodology

Anders Innovations. (12.5.2025). Miksi kanban-menetelmä parantaa tiimin tuottavuutta? https://anders.com/post/miksi-kanban-menetelma-parantaa-tiimin-tuottavuutta

Aseprite. (n.d.). Files. Haettu 27.03.2025 osoitteesta https://www.aseprite.org/docs/files/#aseprite

BIMM University. (23.5.2024). The Art of World-Building in Game Design. https://blog.bimm.co.uk/the-art-of-world-building-in-game-design

Bramble, R. (10.5.2023) The Seven Stages of Game Development. https://gamemaker.io/en/blog/stages-of-game-development#testing

Brazie, A. (n.d.). Video Game Mechanics: A Beginner's Guide (with Examples). https://gamedesignskills.com/game-design/video-game-mechanics/#what-are-video-game-mechanics

Bromberg, M. (12.9.2024). A message to our community: Unity is canceling the Runtime Fee. https://unity.com/blog/unity-is-canceling-the-runtime-fee

Cuaycong, A. (21.5.2024). Good Game Design: The Importance of Sound Design and Music. https://megacatstudios.com/blogs/game-development/good-game-design-importance-sound-design-music?srsltid=AfmBOoqniTD03Di7SPLZ62xLY2RJEn9J_VDz6HrQN4v39qVWOpOktu-x

Denisyuk, Y. (12.3.2024) What are the stages of game development? https://pinglestudio.com/blog/full-cycle-development/game-development-stages

Estevez, C. (29.10.2015) Project planning for solo game developer. https://hacknplan.com/project-planning-for-solo-game-developers/

Haas, J. (2014). A History of the Unity Game Engine. [Väitöskirja, Worcester Polytechnic Institute]. WPI digital repository https://digital.wpi.edu/concern/student_works/tx31qh96p?locale=en

Kramarzewski, A. & De Nucci, E. (2018). Practical Game Design: A modern and comprehensive guide to video game design. Packt Publishing

Martynenko, E. (06.12.2022). The Pros and Cons of Unity Game Engine. Haettu 21.02.2025. osoitteesta https://pinglestudio.com/blog/full-cycle-development/pros-and-cons-of-unity-game-engine

Nuclino. (n.d.). Video Game Development Process. Haettu 28.03.2025 osoitteesta https://www.nuclino.com/articles/video-game-development-process

Rehkopf, M. (n.d.). Scrum Sprints: Everything You Need to Know. https://www.atlassian.com/agile/scrum/sprints

RocketBrush Studio. (28.7.2024a). Godot Engine vs Unity: Which one suits you the best in 2025. RocketBrush Studio Blog. https://rocketbrush.com/blog/godot-vs-unity

RocketBrush Studio. (17.8.2024b). Unity vs Unreal Engine: Which one suits you the best in 2025. RocketBrush Studio Blog. https://rocketbrush.com/blog/unity-vs-unreal-engine-which-one-should-you-choose-in-2024

Scrum. (n.d.). What is Scrum? https://www.scrum.org/resources/what-scrum-module

Shah, M. (16.1.2025). Unity Gaming Engine: A Case Study in Platform Dominance https://shahmm.medium.com/unity-gaming-engine-a-case-study-in-platform-dominance-962e919b76e6

Schell, J. (2008). The Art of Game Design: A Book of Lenses. Morgan Kaufmann Publishers.

Streamline Studios. (25.4.2024). Agile Methodologies in Video Game Development. https://www.streamline-studios.com/post/agile-methodologies-in-video-game-development#EN

Unity Technologies. (2025a). Introduction to GameObjects. Haettu 28.02.2025. https://docs.unity3d.com/Manual/GameObjects.html

Unity Technologies. (2025b). Introduction to scenes. https://docs.unity3d.com/6000.1/Documentation/Manual/CreatingScenes.html

Unity Technologies. (31.01.2023). Introduction to Tilemaps. Haettu 22.05.2025 osoitteesta https://learn.unity.com/tutorial/introduction-to-tilemaps#

Unity Technologies (n.d.). Plans and pricing. https://unity.com/products

Unity Technologies. (2025c). System requirements for Unity 6. https://docs.unity3d.com/Manual/system-requirements.html