HAMK
Hämeen ammatti-
korkeakoulu

2D-pelimoottorin toteutus Monogame-ohjelmistokehyksellä

Ammattikorkeakoulututkinnon opinnäytetyö
Tieto- ja viestintätekniikka, insinööri (AMK)
Kevät, 2024
Ville Keituri

HAMK
Hämeen ammatti-
korkeakoulu

Tieto- ja viestintätekniikka Tiivistelmä
Tekijä Ville Keituri Vuosi 2024
Työn nimi 2D-pelimoottorin toteutus Monogame-ohjelmistokehyksellä
Ohjaaja Petri Kuittinen

Videopelin toteuttaminen modernein keinoin tarkoittaa usein valmiin pelimoottorin, tai pelimoottoriohjelmistokehyksen käyttöä peliprojektissa. Valmiiden pelimoottorien käyttö pelin toteuttamisessa on viime vuosina kasvattanut suosiotaan.

Tämä opinnäytetyö pyrki tutkimaan Monogame-ohjelmistokehyksen soveltuvuutta 2D-pelimoottorin toteuttamisessa. Vertailua muihin videopelin toteutustapoihin tehtiin yleisimpien pelimoottoreiden ja ohjelmistokehysten osalta, sekä pohdittiin eri toteutustapojen etuja sekä haittoja. Pelimoottorin toteutuksesta haettiin tietoa pelimoottorin suunnitteluun erikoistuvista kirjoista, erilaisista dokumentaatiosta sekä aiheeseen liittyvistä tutkimuksista.

Tutkinnan soveltava osuus keskittyi 2D-pelimoottorin toteuttamiseen Monogame-ohjelmistokehyksellä. Tavoite oli toteuttaa ydinkomponenteiltaan tyypillinen 2D-videopelimoottori hyödyntäen entity component system arkkitehtuuria. Konseptitodisteena pelimoottorilla toteutetaan yksinkertainen 2D-tasohyppelypeli.

Opinnäytetyö osoittaa, että yksinkertaisenkin pelimoottorin toteuttaminen on suhteellisen monimutkainen prosessi. Ohjelmistokehyksen valinta pelimoottorin toteuttamiseen antaa lähes täyden vapauden toteuttaa pelimoottorin sekä pelin haluamallaan tavalla, vaikkakin toteutustapa vaatii kohtuullisen paljon pohjatyötä ennen kuin pelimoottorilla voidaan toteuttaa mitään mikä muistuttaisi edes etäisesti peliä. Monogame-ohjelmistokehys on oiva vaihtoehto nimenomaan 2D-pelin toteuttamiseen kaupallisten pelimoottoreiden sijaan. Monogame-ohjelmistokehyksen heikot puolet liittyvät lähinnä pelimateriaalien käsittelyn puutteisiin sekä heikkoon dokumentaation tasoon. Opinnäytetyötä varten toteutettu pelimoottori syvensi omaa käsitystä pelimoottoreiden monimutkaisuudesta.

Avainsanat Pelimoottori, Monogame, Sovelluskehitys
Sivut 48 sivua

HAMK
Hämeen ammatti-
korkeakoulu

Information and Communication Technology
Author Ville Keituri
Subject 2D game engine with Monogame Framework
Supervisor Petri Kuittinen

Abstract
Year 2024

Implementing a video game in modern ways often means using a ready-made game engine or game engine framework in the game project. The use of ready-made game engines in game development has been increasingly popular in recent years.

This thesis aimed to investigate the suitability of the Monogame framework for implementing a 2D game engine. A comparison was made with other methods of video game implementation, focusing on the most common game engines and software frameworks, and discussing the advantages and disadvantages of different implementation methods. Information about game engine implementation was sought from specialized books on game engine design, various documentation, and related research.

The practical part of the study focused on implementing a 2D game engine using the Monogame framework. The goal was to implement a typical 2D video game engine using the entity component system architecture. As a proof of concept, a simple 2D platformer game was implemented with the game engine.

The thesis demonstrates that the implementation of even a simple game engine is a relatively complex process. Choosing a software framework for implementing the game engine provides almost complete freedom to implement the game engine and the game in the desired way, although the implementation method requires a reasonable amount of groundwork before anything resembling a game can be implemented with the game engine. The Monogame software framework is an excellent alternative specifically for implementing 2D games instead of using commercial game engines. The weaknesses of the Monogame framework mainly relate to deficiencies in handling game materials and the poor level of documentation. The game engine implemented for the thesis deepened my understanding of the complexity of game engines.

Keywords Game engine, Monogame, software development
Pages 48 pages

Sisällys

Kuvat, taulukot ja kaavat

Sanasto

AAA-peli	Suuren kehitys- ja markkinointibudjetin peli
Indie-peli	Yksityishenkilöiden tai pienten ryhmien peli
DLL	Dynamic-link library
Metadata	Kuvailevaa tietoa tiedostosta
MIT-lisenssi	Avoin ohjelmistolisenssi
ECS	Entity Component System
SDL	Simple DirectMedia Layer
Ohjelmistokehys	Alusta, joka tarjoaa perustan ohjelmistosovellusten kehittämiseen

1 Johdanto

Aloin harjoittelemaan ohjelmointia tekemällä pienimuotoisia pelejä Unity-pelimoottorilla joitakin vuosia sitten. Veljeni tuotti äänet sekä grafiikan, ja minä hoidin ohjelmoinnin. Pidin pelituotannon haastavuudesta sekä luovuuden tunteesta. Opin paljon ohjelmoinnista, sekä samalla sain olla tekemässä jotain, minkä parissa olin lapsesta asti kuluttanut paljon aikaa. Tajusin kuitenkin, että Unityn kehittäjäystävällinen rajapinta kätkee allensa paljon kompleksisuutta. Tästä heräsi halu tutkia lisää mitä oman pelimoottorin toteuttaminen todellisuudessa vaatii.

Tyypillinen tapa toteuttaa peli on käyttää jotain valmista pelimoottoria. Vaihtoehtoinen toteutustapa pelille on käyttää pelimoottorin sijaan ohjelmistokehystä. Pelien ja pelimoottoreiden kehitykseen suunnitellut ohjelmistokehykset tarjoavat kokoelman ohjelmistokirjastoja kehittäjän käytettäväksi, joiden avulla pelikehittäjät voivat toteuttaa projektinsa haluamallaan tavalla. Tähän tarkoitukseen soveltuvia ohjelmistokehyksiä on useita.

Tämän opinnäytetyön tarkoitus on tutkia, miten Monogame-ohjelmistokehys soveltuu 2D-pelimoottorin toteuttamiseen. Opinnäytetyön edetessä pyritään syventämään tietämystä pelimoottorista käsitteenä, sekä tutkitaan olemassa olevia yleisimmin käytettyjä pelimoottoreita ja ohjelmistokehyksiä. Tutkinta siirtyy pelimoottoriarkkitehtuurin tyypillisiin pelimoottorin ydinkomponentteihin, sekä entiteettien ja komponenttien hallintaan tarkoitettuun ohjelmistoarkkitehtuurisiin ratkaisuihin. Teoriaosuuden jälkeen suunnitellaan 2D-pelimoottori hyödyntäen .NET-komponenttikirjastoa sekä Monogame-ohjelmistokehystä.

Opinnäytetyön soveltava osuus keskittyy teoriaosuudessa läpikäytyjen pelimoottoriarkkitehtuurin tyypillisien komponenttien toteutukseen, sekä esimerkkipelin toteuttamiseen toteutetulla pelimoottorilla. Opinnäytetyön tutkiva ja soveltava osuus on rajattu kaksiulotteisen pelimoottorin tutkimiseen ja toteutukseen. Soveltavan osuuden ulkopuolelle jää moninpelitoimintojen toteutus, pelimoottorin dokumentointi, käyttäjätestaus sekä suorituskyvyn mittaaminen. Soveltavan osuuden tavoite on luoda datakeskeinen helposti laajennettava 2D-pelimoottori.

2 Pelimoottori

Termi ”pelimoottori” syntyi 1990-luvun puolivälissä Id softwaren Doom pelin kaltaisten suosittujen ensimmäisen persoonan ammuntapelien takia. Doomin kaltaisten hyvin suunniteltujen pelien tapa erotella ohjelmiston ydinkomponentit ja pelaajan käyttäjäkokemusta ohjaavista asioista. (Gregory, 2014, s. 11)

Pelimoottori on tietokoneohjelmisto mikä on suunniteltu videopelien toteutukseen sekä kehitykseen. Pelikehittäjän käyttävät pelimoottoreita toteuttaakseen videopelejä konsoleille, mobiililaitteille sekä tietokoneille. Pelimoottorit virtaviivaistavat videopelin luomisen prosessin pelin eri ydinkomponenttien osalta. Pelin ydinkomponentteja ovat muun muassa grafiikan renderöinti, hahmojen liikkeiden animointi, fysiikan lakien simulointi, ääniefektien toistaminen sekä pelihahmojen tekoäly. (Gupta, 2023)

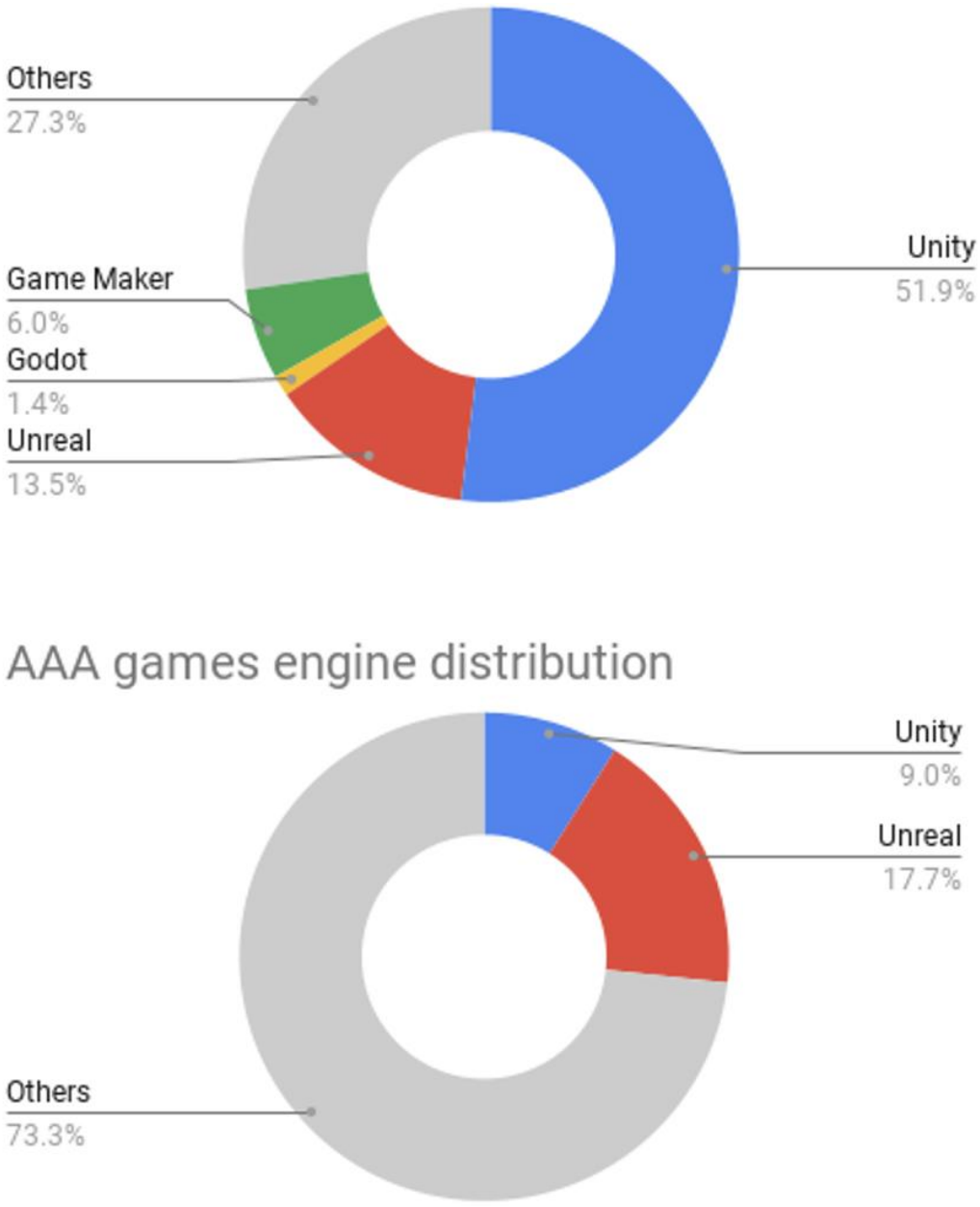
Nykypäivän modulaarisesti rakennetuissa peleissä pelin moottorilla tarkoitetaan simulaatiokoodia suorittavaa komponenttikokoelmaa, joka ei suoraan määrittele pelin pelilogiikkaa tai peliympäristöä. Moottori sisältää komponentit syötteen käsittelyyn, kuvan piirtämiseen (3D ja 2D-renderöinti), sekä yleiseen fysiikan ja dynamiikan mallinukseen. (Lewis & Jacobson, 2002)

2.1 Käytetyimmät pelimoottorit

Pelimoottoreiden kirjo ovat yhtä monipuolinen kuin pelien tyylilajit ja alustat, joille niillä luodaan pelejä. Studio, joka pyrkii luomaan kerrontapainotteisen 2D-tasohyppelypelin, voisi hyödyntää esimerkiksi Unity-pelimoottoria pelin toteutuksessa sen monikäyttöisyyden ja laajan materiaalivalikoiman vuoksi. Toisaalta se, joka työskentelee graafisesti vaativan pelin parissa, saattaisi suosia Unreal Engineä sen korkean tarkkuuden renderöintiominaisuuksien vuoksi. (Game Ace Studio, 2023) Mikäli studio priorisoi joustavuutta sekä kustannustehokkuutta, voi studio käyttää projektissaan esimerkiksi avoimen lähdekoodin Godot-pelimoottoria (Morrow, 2023).

Suurimmilla peliyrityksillä on resursseja, aikaa sekä liiketoiminnallisia syitä luoda yrityksen sisäiseen käyttöön tarkoitettu pelimoottori. Indie peliyrityksissä työskentelevistä kehittäjistä vähemmän kuin 5 % käyttää studion sisäisesti rakentamia pelimoottoreita. (SlashData, 2022, s. 51)

Kuva 1 Pelimoottoreiden käyttöaste (Milenovic, 2023)

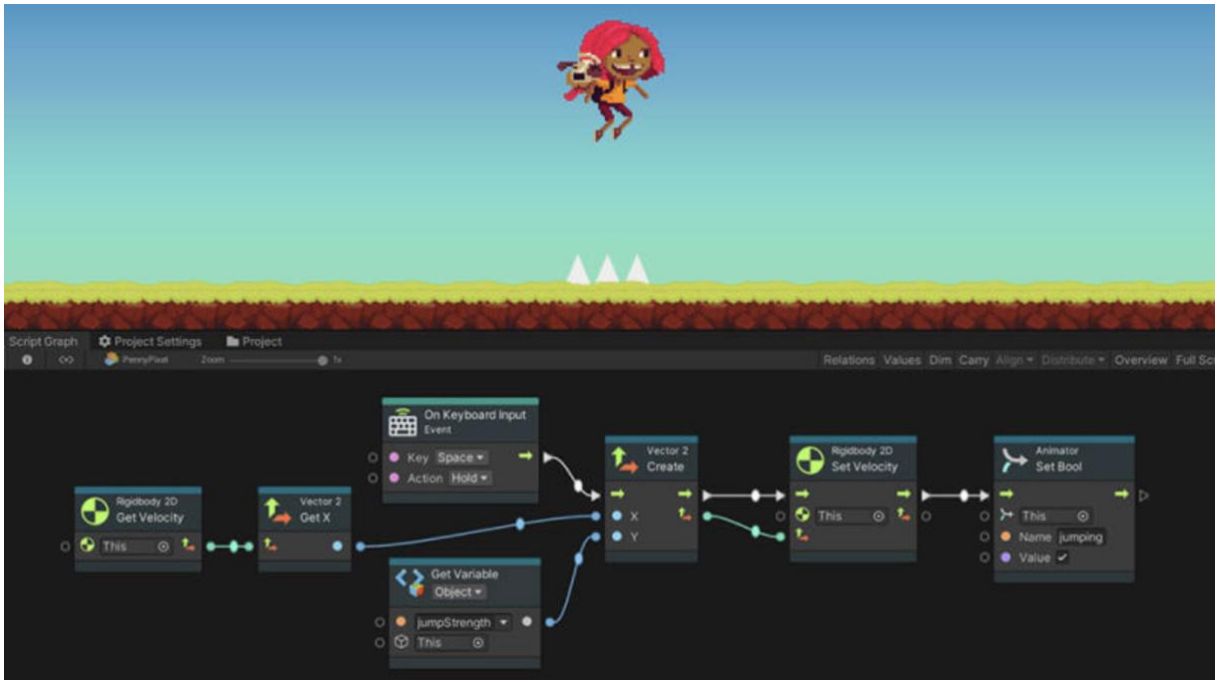


2.1.1 Unity

Unity Technologiesin kehittämä Unity-pelimoottori on vuodesta 2005 eteenpäin kehitetty 2D ja 3D-pelimoottori. Unity vahvuuksiin kuuluu selkeä arkkitehtuuri, tuki alustariippumattomalle pelikehitykselle, laaja materiaalivalikoima, sekä useat valmiit työkalut ja laajennukset auttamaan pelikehittäjää työssään. Näiden asioiden takia Unity on yksi maailman suosituimmista pelimoottoreista, erityisesti indie-pelikehittäjien keskuudessa. (Zenva, 2023)

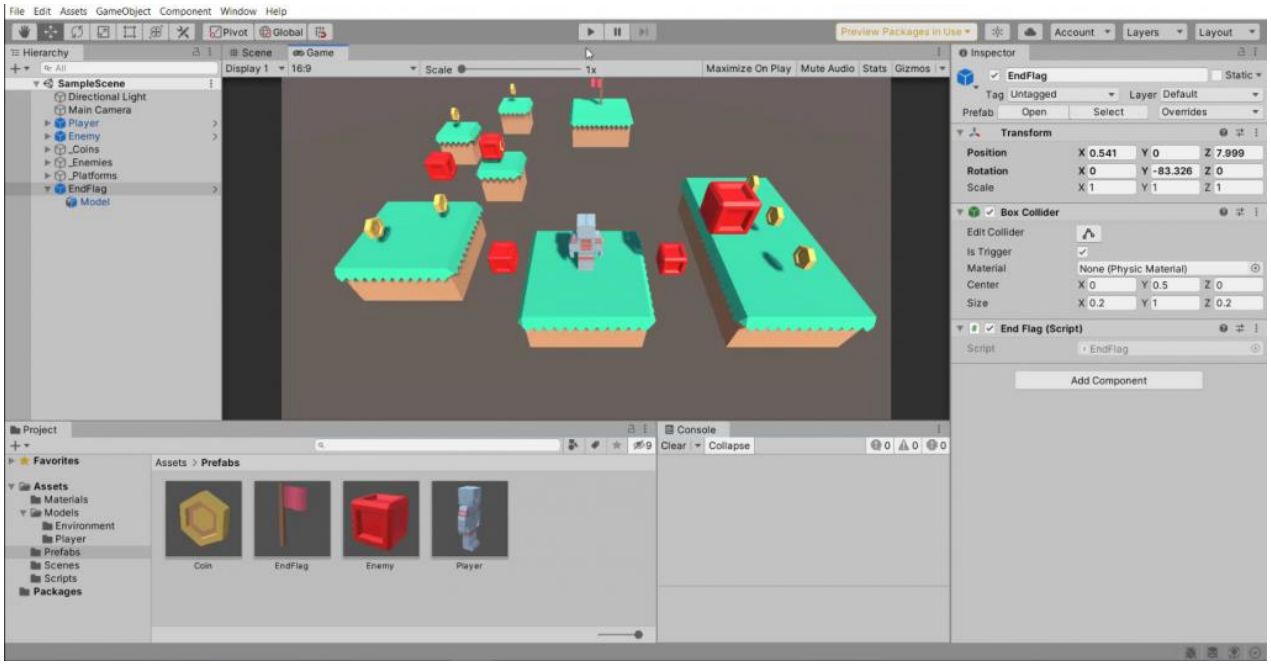
Unity on ilmainen ladata ja käyttää (tietyin edellytyksin), joten se sopii hyvin aloitteleville pelikehittäjille. Unity virtaviivaistaa 3D-pelikehitystä, sekä Unity-pelimoottorin tukema muistihallittu C# kieli helpottaa pelikehitystä verrattuna pelimoottoreiden ohjelmointiin tyypillisesti käytettyyn C++ kieleen verrattuna. Ohjelmointia ei välttämättä tarvitse tehdä lainkaan, kun Unity julkaisi vuonna 2020 visuaalisen skriptityökalun, mikä mahdollistaa pelilogiikan luonnin noodipohjaisella editorilla ilman varsinaista ohjelmointia. (Dealessandri, 2020b)

Kuva 2 Unityn visuaalinen skriptityökalu (Unity Technologies, n.d.b)



Peliobjekti on Unity-pelimoottorin tärkein käsite. Kaikki pelin entiteetit kuten hahmot, valot, kamera sekä erikoistehosteet ovat peliobjekteja. Peliobjekti on abstrakti käsite, millä ei itsessään ole mitään toimintoa, vaan pelikehittäjän tehtävä on luoda peliobjektille komponentteja, mitkä määrittävät peliobjektin toiminnallisuuksia. Unity sisältää paljon valmiita komponentteja, sekä Unity tarjoaa mahdollisuuden ohjelmoida omia komponentteja. (Unity Technologies, 2021)

Kuva 3 Unity pelimoottorin editori. (Zenva, 2023)



Unityllä tuotettuihin tunnettuihin pelisarjoihin lukeutuu muun muassa Pillars of Eternity sekä Ori and the blind forest pelisarjat, Pokemon GO sekä Heartstone. (Drake J. , 2023a)

Unity monesta vahvuudesta huolimatta pelimoottoria ei usein käytetä laajan skaalan AAA-pelien toteutukseen. Unityn alottelijaystävällisyyden kääntöpuoli on se, että pelimoottorin toimintoja on vaikea tai jopa mahdoton optimoida AAA-pelien tarpeisiin. (Dealessandri, 2020b) Hyvän sovellusarkkitehtuurin toteuttaminen Unity-pelimoottorilla toteutettavaan peliin saattaa olla haasteellista, sillä Unity-kehitysympäristö itsessään ei kannusta kehittäjää käyttämään hyviä toteutustapoja. (Neurosys, 2022)

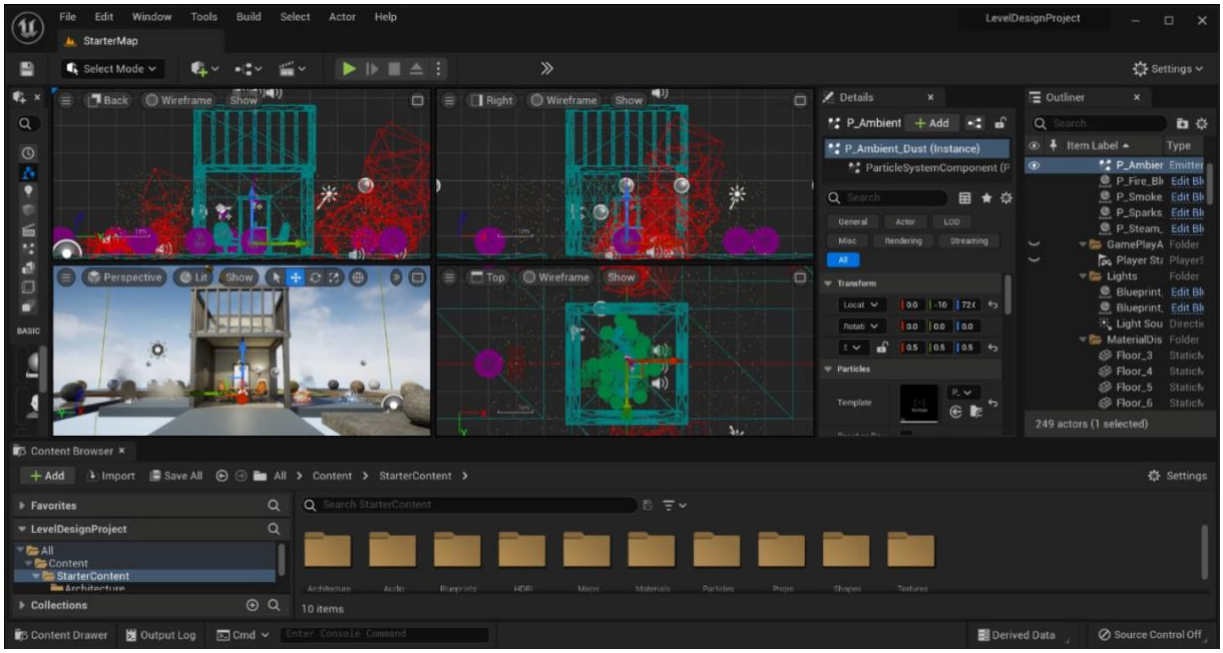
2.1.2 Unreal Engine

Unreal Engine on Epic Gamesin omistama edistynyt 3D-pelimoottori. Pelimoottorin ensimmäisen versio julkaistiin vuonna 1998 ensimmäisen persoonan ammuntapelin Unreal yhteydessä. Pelimoottori sisälsi karttaeditorin, UnrealEd, mikä tarjosi pelaajille ainutlaatuisen mahdollisuuden luoda omia tasoja pelattavaksi. (Zippia, 2024) Nykyään Unreal-pelimoottori on tunnettu korkeatasoisesta grafiikan toistosta, sekä nopeasta suorituskyvystä. Tämän vuoksi Unreal Engineä voidaan käyttää muun muassa erilaisissa viihdeteollisuuden toteutuksissa, muun muassa AAA-luokan peleissä, 3D-visualisoinnissa, arkkitehtuurillisessa videotuotannossa sekä simuloinneissa. (Erolin, n.d)

Kuten Unityn peliobjekti, Unreal Enginessä perusentieetti on Actor. Actor instanssin entiteettiin voidaan liittää komponentteja mitkä tarjoavat entiteeteille jonkin toiminnallisuuden. Actor on kantaluokka, joista on periytetty aliluokkia, muun muassa Pawn sekä Character. Pawn edustaa pelaajan tai tietokoneen ohjaaman entiteetin fyysistä muotoa pelimaalimassa. Pawn määrittelee muun muassa miltä entiteetti visuaalisesti näyttää pelimaailmassa sekä minkälainen vuorovaikutus sillä on muihin pelimaailman entiteeteihin. Character on Pawn luokan alityyppi, minkä avulla voidaan määritellä minkälaisessa ympäristössä entiteetti tulee toimimaan pelimaailmassa. (Epic Games, 2024)

Unreal Enginen vahvoihin puoliin kuuluu pitkälle kehitetty jälkikäsittely- sekä animointiteknologia minkä avulla artistit kykenevät luomaan erittäin elävän kaltaisia hahmoja. Pelikehittäjän voivat myös hyödyntää Unreal Enginen tarkasti todellisuutta mukailevaa fysiikan mallinnusta vaativissa peliprojekteissaan. Pelimoottorilla voi myös luoda pelilogiikkaa noodipohjaisella visuaalisella skriptityökalulla (Blueprint). Näiden edistyneiden työkalujen ja teknologioiden varjopuolena on Unreal Enginen jyrkkä oppimiskynnys, pelimoottorin tehokas käyttö vaatii pitkäjäteistä opettelua. Pelimoottorilla tuotettujen pelien resurssivaativa luonne voi turhaan kasvattaa pelin tiedostokokoa, sekä siten pidentää pelin latausaikoja. Pelin tuottosidonnaiset lisensointimaksut voivat myös olla korkeat. (Eventyr, 2023)

Kuva 4 Unreal Engine -pelimoottorin editorinäkymä (Epic Games, 2024)



Tunnettuja pelejä mitkä ovat toteutettu Unreal Enginellä on muun muassa ammuntapeli Fortnite, toimintaseikkailupeli Hellblade: Senua’s sacrifice sekä toimintaroolipeli Borderlands 3. (Drake J. , 2023b)

2.1.3 Godot

Godot on avoimen lähdekoodin yleiskäyttöinen 2D ja 3D-pelimoottori. Godot kehitettiin alun perin Argentiinalaisen pelistudion sisäiseen käyttöön vuonna 2001. Juan Linietsky ja Ariel Manzur julkaisivat Godot-pelimoottorin lähdekoodin yleiseen käyttöön vuonna 2014. (Godot, n.d) Godot-pelimoottori on lisensoitu MIT-lisenssillä, joten sen käyttö projektissa on täysin maksutonta, eikä pelimoottorilla tehdystä pelistä tarvitse maksaa lisensointi tai rojaltimaksuja. Avoin lähdekoodi antaa pelikehittäjille mahdollisuuden muuttaa pelimoottorin toimintaa omiin tarpeisiinsa sopivaksi. (Bradfield, 2018, ss. 8−9)

Godot-pelimoottori tarjoaa pelimoottorille tyypilliset toiminnallisuudet kuten animointityökalut, ääniefektien käsittelytyökalut, fysiikkamoottorin, pelieditorin, sekä moninpelivalmiudet. Godot on alustariippumaton pelien kehitysympäristö. (Night Quest Games, 2023) Kaupallisten pelimoottoreiden tapaan Godot sisältää visuaalisen skriptityökalun millä pelilogiikkaa voidaan luoda ilman ohjelmointikielen käyttöä. (Ahamed, 2023)

Kuva 5 Godot-pelimoottorin editori (Godot, n.d)

Suljetun lähdekoodin pelimoottoreihin kuten Unityyn tai Unreal Engineen verrattuna Godot-pelimoottori ei tarjoa yhtä laajaa valikoimaan erilaisia toiminnallisuuksia. (Night Quest Games, 2023). Pelimoottori soveltuu myös heikosti monimutkaiseen 3D pelien toteutukseen, sekä dokumentoinnissa on puutteita. (Ahamed, 2023)

Tunnettuja Godot-pelimootorilla julkaistuja pelejä ovat muun muassa 3D-tasohyppelypeli Sonic Colors Ultimate, 2D-kauhupeli Endoparasitic sekä 2D-roguelike peli Brotato. (Darling, 2023)

2.2 Ohjelmistokehykset

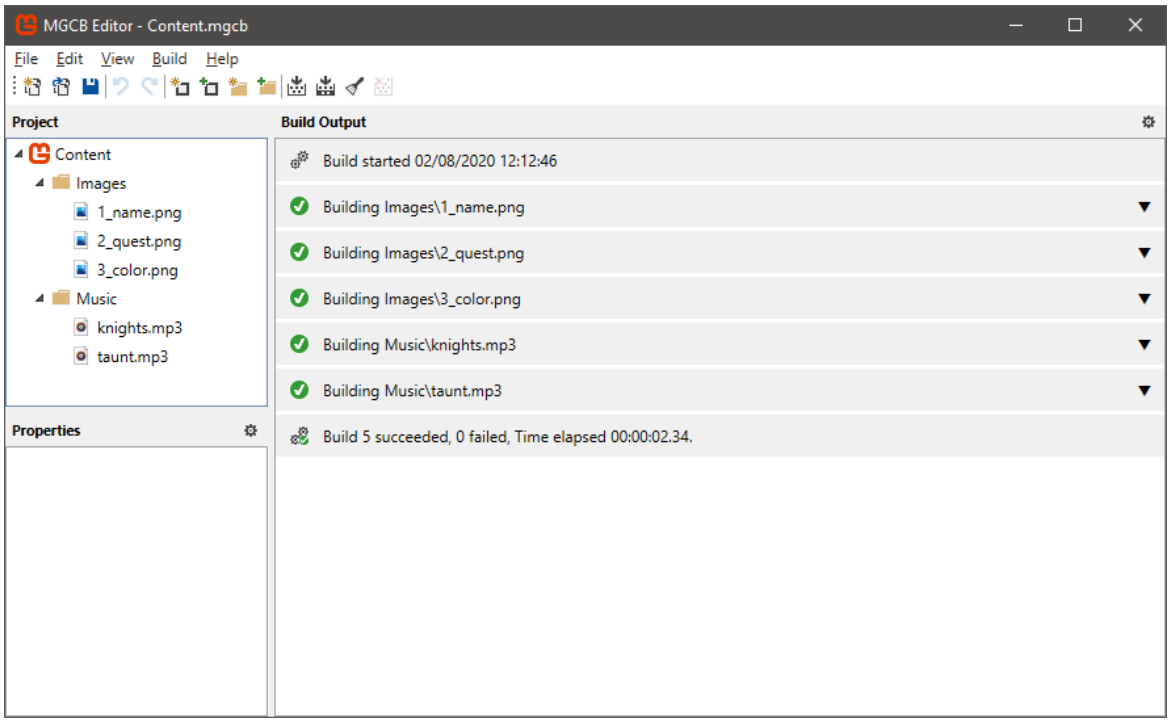
Ohjelmistokirjasto on kokoelma koodia sekä dataa. Ohjelmistokirjastot ovat usein binääritiedostoja, tai DLL-sovelluslaajennustiedostoja. Ohjelmistokirjaston tehtävä on suorittaa jokin toiminto, esimerkiksi syötteen vastaanottaminen käyttäjältä. Ohjelmistokehys, samoin kuin pelimoottori, on kokoelma erilaisia ohjelmistokirjastoja. Ohjelmistokehys tarjoaa yleiskäyttöisen minimaalisen toiminnallisuuden, minkä käyttäjä voi halutessaan valikoivasti korvata tai laajentaa, tuottaakseen sovelluskohtaisen ohjelmiston. Toisin kuin ohjelmistokehys, pelimoottori tarjoaa käyttäjälle paljon laajemman kokoelman valmiiksi rakennettuja ratkaisuja. (Dragonfly, n.d; GameFromScratch, 2015)

Seuraavissa luvuissa tarkastellaan muutamia pelimoottorin ja pelien kehitykseen käytettyjä ohjelmistokehyksiä.

2.2.1 Monogame

Monogame on avoimen lähdekoodin ohjelmistokehys, mikä jäljittelee Microsoftin vuonna 2006 julkaisemaa XNA ohjelmistokehystä. Microsoft lopetti XNA-ohjelmistokehyksen tukemisen vuonna 2013, jolloin Monogame sai alkunsa XNA:n pelikehittäjäyhteisön toimesta. Monogame ohjelmistokehys on kirjoitettu C#-ohjelmointikielellä. Monogame abstrahoi alustariippuvuuden grafiikan, äänen ja muiden pelin suorittamiseen tarvittavien teknologioiden osalta, joten kirjoitettu ohjelmakoodi voidaan kääntää usealle eri alustalle suoritettavaan muotoon. (Jackson, n.d-b) Monogame sisältää Content Pipeline-ohjelmiston pelisisällön kuten fonttien, ääniefektien tai bittikarttakuvien lataamiseen. Content Pipeline serialisoi ja pakkaa pelisisällön jokaiselle Monogamen tukemalle alustalle haluttuun muotoon. (Jackson, n.d-a)

Kuva 6 Monogame Content Pipeline editori (Monogame, n.d-a)



Monogamen vahvuudet painottuvat sen alustariippumattomuuteen, kustomoitavuuteen, sekä pelin nopeaan kehityssykliin. Monogame tukee nopeaa iterointia erilaisten ominaisuuksien kautta, kuten resurssien ajonaikaista uudelleenlatausta ja reaaliaikaisia koodipäivityksiä. Tämä mahdollistaa kehittäjien nähdä muutosten tulokset välittömästi ilman koko projektin uudelleenkääntämistä, mikä nopeuttaa kehitysprosessia merkittävästi. Monogame-ohjelmistokehys on hyvä valinta 2D, sekä vähemmän resurssi-intensiivisille 3D peleille. 3D pelin toteuttaminen on ylipäätään huomattavasti haastavampaa, ja koska Monogame on hyvin pelkistetty ohjelmistokehys, se ei tarjoa pelimoottorille tyypillisiä ydinkomponentteja vakiona vaan kehittäjän on ohjelmoitava ne itse. Tämä tarkoittaa, että Monogame-ohjelmistokehys sopii pelikehittäjille, jotka kaipaavat kontrollia pelimoottorin toimintaan sekä pelin toteutukseen. (Dealessandri, 2020a)

Monogame-ohjelmistokehyksellä on tuotettu muun muassa 2D indie viljelysimulaatio Stardew Valley (2016), metrodivania tyyppinen toimintaseikkailupeli Axiom Verge (2015) sekä toimintaroolipeli Bastion (2011). (Monogame, n.d-b)

2.2.2 SDL

SDL on matalan tason alustariippumaton ohjelmistokirjasto. SDL abstrahoi tietokoneen laitteistokohtaisen tietojenkäsittelyn, ja tarjoaa rajapinnan äänen, syötejärjestelmien sekä grafiikkaohjaimien käsittelyyn. SDL-ohjelmistokirjasto on kirjoitettu C-ohjelmointikielellä, sekä

se tukee C++ ohjelmointikieltä. (SDL, n.d) SDL:lle on myös luotu tuki useille muille ohjelmointikielille kuten Go, C# ja Python. (Loach, 2023)

2.2.3 Pixi.js

PixiJS on 2D-grafiikan alustariippumaton renderöintikirjasto, mikä helpottaa web-selain pohjaisten sovellusten ja pelien toteutuksen. Pixi Js luo ohjelmistokerroksen WebGL:n (Web Graphics Library) päälle, mikä auttaa graafisesti rikkaan sisällön tuottamista selainsovelluksissa. PixiJS tarjoaa käyttäjälle työkaluja muun muassa tekstin ja syötteen käsittelyyn, tiedostojen lataukseen ja käsittelyyn sekä tekstuurien lataukseen. (PixiJS, n.d)

2.3 Pelimoottori vai ohjelmistokehys

Valmiin pelimoottorin (Unity, Unreal Engine tai muu vastaava) käyttö peliprojektissa, etenkin 3D-pelin kohdalla, säästää huomattavasti kehitysaikaa, ja antaa kehittäjälle vapauden keskittyä pelin muiden osa-alueiden kehittämiseen. Kehittäjän itse rakentaessa pelimoottorin esimerkiksi ohjelmistokehyksen avulla, kehittäjä saa kuitenkin täyden kontrollin pelimoottorin eri osa-alueiden toimintaan. (Dragonfly, n.d) Pelimoottori sisältää paljon logiikkaa ohjelmoitavan rajapintansa alla, mitä kehittäjä ei valmista pelimoottoria käyttämällä välttämättä näe. Pelimoottorin rakentaminen on hyvä keino oppia, miten pelimoottori todellisuudessa toimii.

Yksityisomisteiset pelimoottorit sisältävät usein jonkinlaisen maksujärjestelmän, missä pelimoottoria käyttävän yrityksen on maksettava pelimoottorin rakentaneelle yritykselle jonkin suuruinen korvaus pelimoottorin käytöstä. Maksu saattaa olla kertaluonteinen, kehittäjäkohtainen vuosimaksu, tai pelin myyntituottoihin sidoksissa oleva maksu. Pelimoottoreita tuottavat yksityisomistukselliset yritykset voivat muuttaa lisensointikäytäntöjään, tämä on asia mikä ei koske tyypillisiä avoimen lähdekoodin pelimoottoreita ja ohjelmistokehyksiä, jotka sisältävät jonkin suopean ohjelmistolisenssin. Lisensointikäytäntöjen muutoksesta esimerkkinä toimii paljon negatiivista palautetta saanut vuonna 2023 Unity Technoligies:n pyrkimys muuttaa Unity-pelimoottorin lisensointi asennuskohtaiseksi maksuksi, missä pelikehittäjä on velvollinen maksamaan korvauksen jokaisesta asennuksesta (Orland, 2023). Avoimen lähdekoodin pelimoottorin käyttö, tai oman pelimoottorin rakentaminen ohjelmistokehyksen avulla vapauttaa yrityksen tai kehittäjän näistä maksuista.

3 Tasohyppelypeli

Tasohyppelypelit ovat yksi varhaisimpia videopelilajeja. Ensimmäiset tasohyppelypelit kehitettiin 1980 luvulla. Tämän aikakauden peleistä lähes kaikki pelit olivat kaksiulotteisia pelikonsoleiden ja tietokoneiden teknisten rajoitteiden vuoksi. Tyypillinen tasohyppelypeli koostuu persoonan näkökulmasta kuvatusta pelaajahahmosta, joka liikkuu ja hyppii pelialustoilla. Sivusuuntaisissa tasohyppelypeleissä pelaaja liikkuu kohti pelinäytön toista reunaa. Pelaajat kulkevat eteenpäin keräten esineitä, voittaen vihollisia sekä suorittaen erilaisia tehtäviä, kunnes taso on suoritettu. (Klappenbach, 2021)

Klassisia tasohyppelypelejä on muun muassa Super Mario pelisarja, Sonic The Hedgehog pelisarja, sekä hieman modernimmista peleistä voisi mainita Hollow Knight pelin sekä Ori and the Blind Forest pelisarjan.

Kuva 7 Super mario tasohyppelypeli (Nintendo, n.d)

4 Pelimoottoriarkkitehtuuri

Pelimoottori tulisi olla toteutettu useiden itsenäisten moduulien avulla. Jokainen moduuli on itsenäinen toiminnallinen yksikkö. Modulaarinen pelimoottoriarkkitehtuuri vähentää pelimoottorin monimutkaisuutta sekä lisää sen luotettavuutta. Itsenäisten moduulien testausprosessi on myös yksinkertaisempi. (Zarrand, 2018, s. 79)

4.1 Syötejärjestelmä

Pelimoottorin on pystyttävä ottamaan vastaan ja käsittelemään pelaajan syötteitä erilaisista syötelähteistä. Syötejärjestelmä vastaanottaa tai lukee syötteitä muun muassa hiirestä, näppäimistöstä tai peliohjaimesta. Syötteitä on kahdenlaisia:

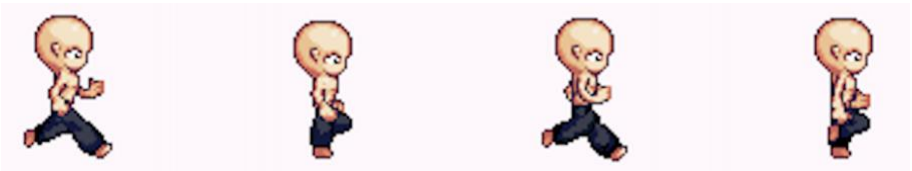
- Syötetapahtuma, pelaajan yksittäinen toimi mikä käynnistää pelimoottorissa jonkin toiminnon
- Syötteen seuranta, pelimoottori aktiivisesti tarkastelee jonkin syötejärjestelmän tilaa

(Villaneuva, 2024)

4.2 Animointijärjestelmä

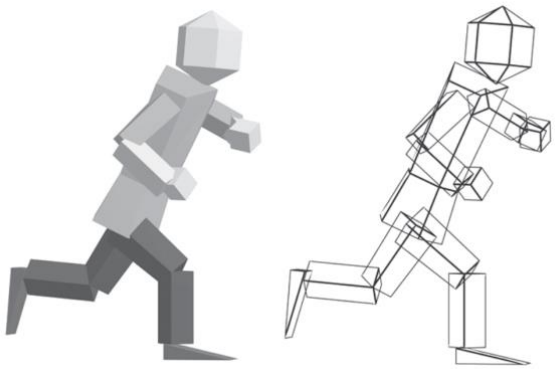
Pelin kaikki entiteetit, jotka eivät ole kiinteitä tai liikkumattomia, voidaan animoida. Sprite animointi on perinteinen 2D animointitekniikka, missä liikkeen illuusio tuotetaan näyttämällä nopeasti peräkkäisiä kuvia. Kuvakehyksen vaihtotaajuus on korkea, jolloin animaatio näyttää sulavalta. Kuvat, joista animaatio koostuu, tuotetaan usein niin että kuvasarjan alku ja loppu voidaan sulavasti yhdistää, jolloin animaatioita on mahdollista toistaa loputtomasti peräkkäin. (Gregory, 2014, s. 492)

Kuva 8 Juoksevan hahmon sprite animaatio. (Dalton, 2018)



Jäykkä hierarkkinen animaatio oli ensimmäisiä 3D-animointitekniikoita, missä animoitava hahmo on mallinnettu useasta jäykästä osasta, esimerkiksi ihmistä mallintava hahmo saattaisi koostua käsistä, sormista, reidestä, säärestä, varpaista, päästä ja vartalosta. Mallin osat ovat hierarkkisesti sidottu toisiinsa, jolloin esimerkiksi reittä liikuttamalla liikkuu myös reiteen liitetty sääri sekä sääreen liitetyt varpaat. (Gregory, 2014, ss. 492-493) (Drake J. , 2023b)

Kuva 9 Jäykkä hierarkinen animaatio (Cranberg, 2009)



4.3 Fysiikan mallinnus

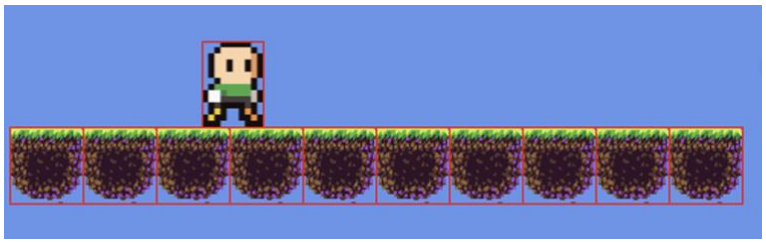
Vastoin todellista maailmaa, virtuaalisessa pelimaailmassa peliobjekteilla ei ole mitään käyttäytymistä, pelikehittäjien tehtävä on luoda se peliobjektille simuloimalla todellisen maailman fysiikkaa jollain tasolla. Fyysinen simulointi pelissä voi tarkoittaa muun muassa kahden esineen yhteentörmäyksen vaikutuksen simulointia, tuhoutuvien esineiden ja rakennusten mallintamista, esineiden käsittelyä, pelihahmojen räsynukke efektejä, tai painovoiman vaikutuksen simulointia. Täydellinen fysiikan simulointi ei tarkoita sitä, että pelin viihdearvo paranee, usein vaikutus saattaa olla päinvastoin. Fyysinen mallinnus luo peliin epävarmuutta, pelissä tarkkojen toimintojen suorittaminen tismalleen samalla tavalla on sitä haastavampaa mitä tarkemmin fysiikan simulointi on pelissä toteutettu. Tarkka fysiikan mallinnus luo pelimoottorin toteutukseen monimutkaisuutta. (Gregory, 2014, ss. 595-600)

4.4 Törmäyksien tarkastelu

Peliobjektien törmäyksien tarkastelu on usein tiukasti liitetty pelin fysiikan mallinnuksen kanssa (Gregory, 2014, s. 595). Fysiikan mallinnuksessa jokainen simuloitava objekti on yleensä yhdistetty yhteen törmäytinobjektiin. Törmäytysobjektin on tietomalli, mikä sisältää

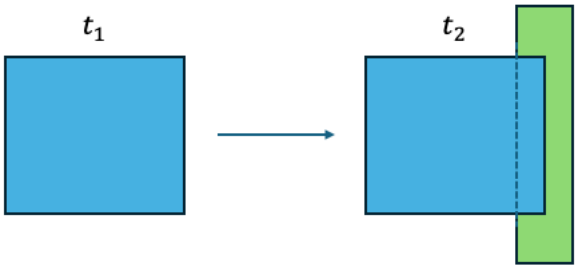
tiedon fysiikan mallinnuksen kohteena olevan objektin muodosta. Törmäytinobjekteja käytetään tarkastelemaan peliobjektien päällekkäisyyksiä, tätä tietoa voidaan käyttää esimerkiksi estämään peliobjekteja liikkumasta toistensa läpi. Törmäytinobjektin muotoina suositaan matemaattisesti ja geometrisesti yksinkertaisimpia mahdollisia muotoja. (Gregory, 2014, ss. 604-606)

Kuva 10 Suorakulmion muotoinen törmäytinobjekti, indikoitu punaisella ääriviivalla.



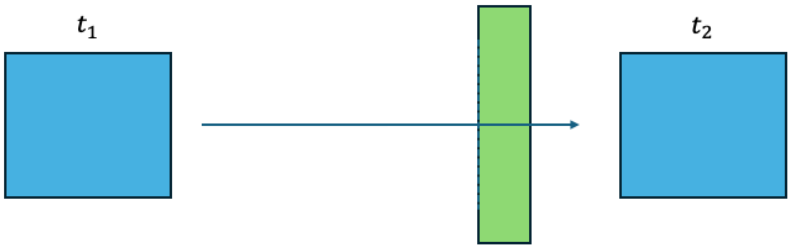
Peliobjektien törmäyksen tarkastelussa käytetään perinteisesti kahta eri menetelmää, törmäytinobjektin päällekkäisyyden tarkastelua tai läpimenon tarkastelua. Päällekkäisyyden tarkastelussa voidaan todeta, onko kahden peliobjektin törmäys jo tapahtunut. (Rich, n.d)

Kuva 11 Päällekkäisyyden tarkastelun havainnekuva



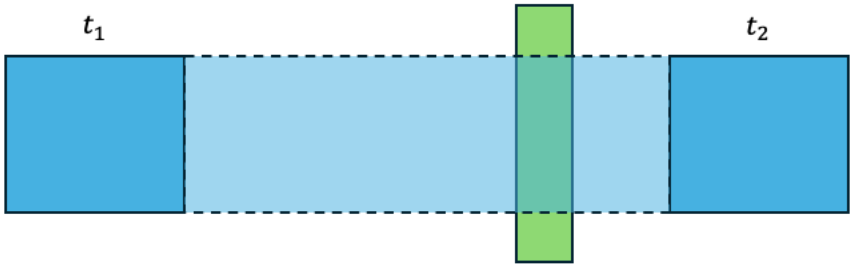
Sinisen ja vihreän peliobjektin päällekkäisyyden tarkastelu ajanhetkillä t_1 ja t_2. Päällekkäisyys ajanhetkellä t_2. Päällekkäisyyden tarkastelu on yleisimmin käytetty tekniikka peleissä, vaikka tämä tekniikka on virhealttiimpi etenkin nopeasti liikkuvien peliobjektien kohdalla, kun objekteilla ei ole päällekkäisyyttä simulaation ajanhetkien kohdalla (Rich, n.d).

Kuva 12 Päällekkäisyyden tarkastelun puute



Päällekkäisyyden tarkastelun sijaan läpimenon tarkastelussa voidaan todeta, tuleeko peliobjektien törmäys tapahtumaan tulevaisuudessa. Törmäytysobjektin muotoa jatketaan sen paikan muutoksen verran liikkeen suuntaan. (Rich, n.d) Läpimenon tarkastelun toteutus on myös monimutkaisempi verrattuna päällekkäisyyden tarkasteluun verrattuna.

Kuva 13 Peliobjektin läpimenon tarkastelu



Peliobjektia jatketaan paikan muutoksen verran liikkeen suuntaan kuvassa 13. Päällekkäisyys voidaan todeta peliobjektista ajanhetkestä t_1 katkoviivoin jatketussa osassa.

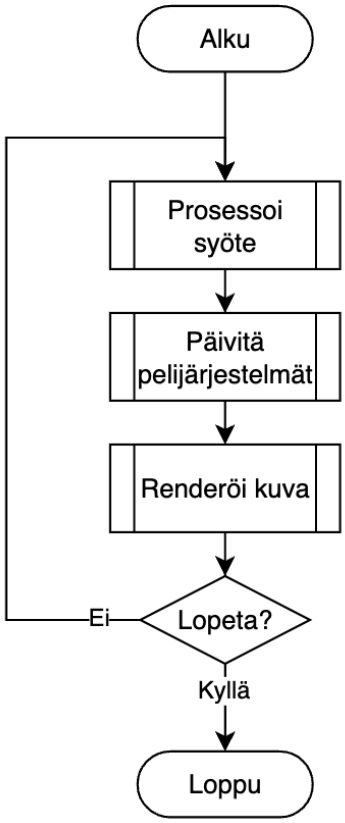
4.5 Resurssien hallinta

Pelit ovat multimediasovelluksia, joten pelimoottorin on kyettävä lataamaan ja käsittelemään useita eri tiedostoja. Peli saattaa käyttää muun muassa bittikarttoja, 3D tahkoverkkoja, ääniraitoja ja tekstitiedostoja. Pelimoottori sisältää usein jonkinlaisen sisäisen resurssimanagerin, minkä tehtävä on ladata tarpeellisia resursseja tietokoneen muistiin, ylläpitää ladattujen tiedostojen elinkaarta ja kirjoittaa koneen levylle tietoa. Resurssimanageri toteutetaan natiivin levyjärjestelmän rajapinnan päälle toteutettavalla pelimoottorikohtaisella rajapinnalla, jolloin pelimoottori voidaan toteuttaa tukemaan useaa eri levyjärjestelmää. (Gregory, 2014, s. 261)

4.6 Pelisilmukka

Pelimoottori koostuu useasta alijärjestelmästä, osaa näistä tutkittiin edellisissä luvuissa. Nämä alijärjestelmät vaativat säännöllistä ylläpitoa pelin ollessa käynnissä. Ylläpitosykli kuitenkin vaihtelee järjestelmästä toiseen, esimerkiksi animaatiot on päivitettävä 30 tai 60 kertaa sekunnissa, fysiikkajärjestelmä saattaa laskea pelin tilaa 120 kertaa sekunnissa, mutta hahmojen tekoälyjärjestelmän on ylläpidettävä kerran tai kaksi sekunnissa. Tätä ylläpitosykliä kutsutaan pelisilmukaksi. (Gregory, 2014, ss. 303-304)

Kuva 14 Yksinkertaistettu havainnekuva pelisilmukasta



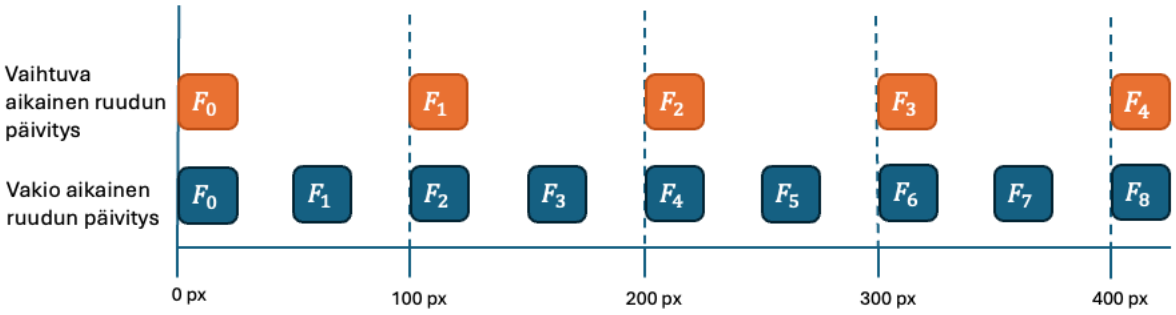
Pelisilmukka sisältää kaiken logiikan mitä yhden ruudunpäivityksen aikana on tapahduttava. Pelisilmukan ruudunpäivityksien välinen aika lähestulkoon aina muuttuva, pelimoottori voidaan kuitenkin toteuttaa tukemaan vain vakioaikaista ruudunpäivitystä.

Vakioaikaisessa ruudunpäivityksessä peli päivittyy jollakin ennalta määrätyllä taajuudella, esimerkiksi 30 ruudunpäivitystä sekunnissa. Tämä tarkoittaa, että pelimoottorilla on $\frac{1}{30}$ sekuntia (~33 ms) aikaa muodostaa seuraava piirrettävä kuva. Aikariippuvaiset alijärjestelmät, kuten animointi tai fysiikan simulointi päivittyvät tällä ennalta määrätyllä taajuudella mikä tarkoittaa, että peli suoritetaan samalla nopeudella jokaisella laitteella.

Kiinteäaikaisen ruudunpäivityksen yksi merkittävä heikkous on se, että mikäli peliä suorittava kone ei kykene suorittamaan peliä tarpeeksi nopeasti, peli toimii näkyvästi hitaammin, koska pelimoottori ei kompensoi järjestelmän hitautta. Tästä syystä tämä tekniikka soveltuu esimerkiksi vanhoille konsoleille, kun pelikehittäjällä on tarkalleen tiedossa, minkälaisella laitteella peliä suoritetaan. Vaihtuva-aikaisessa ruudunpäivityksessä pelimoottori pyrkii kompensoimaan laitteiden suorituskykyeroja käyttämällä aikariippuvaisten järjestelmien laskennoissa hyödyksi ruudunpäivitysten välisen ajan kestoa. (Foltz, 2011)

Kuvassa 15 yritetään havainnollistaa vaihtuva-aikaisen sekä kiinteäaikaisen ruudunpäivityksen eroa. Molemmat objektit on ohjelmoitu liikkumaan 100 pikseliä sekunnissa oikealle. Suorittava laite kuitenkin pystyy päivittämään ruutua vain puolella siitä ruudunpäivityssyklistä mitä pelimoottorin alijärjestelmät olettavat, jolloin vakio aikaisen ruudunpäivityksen objekti liikkuu lyhyemmän matkan jokaisen ruudunpäivityksen välillä.

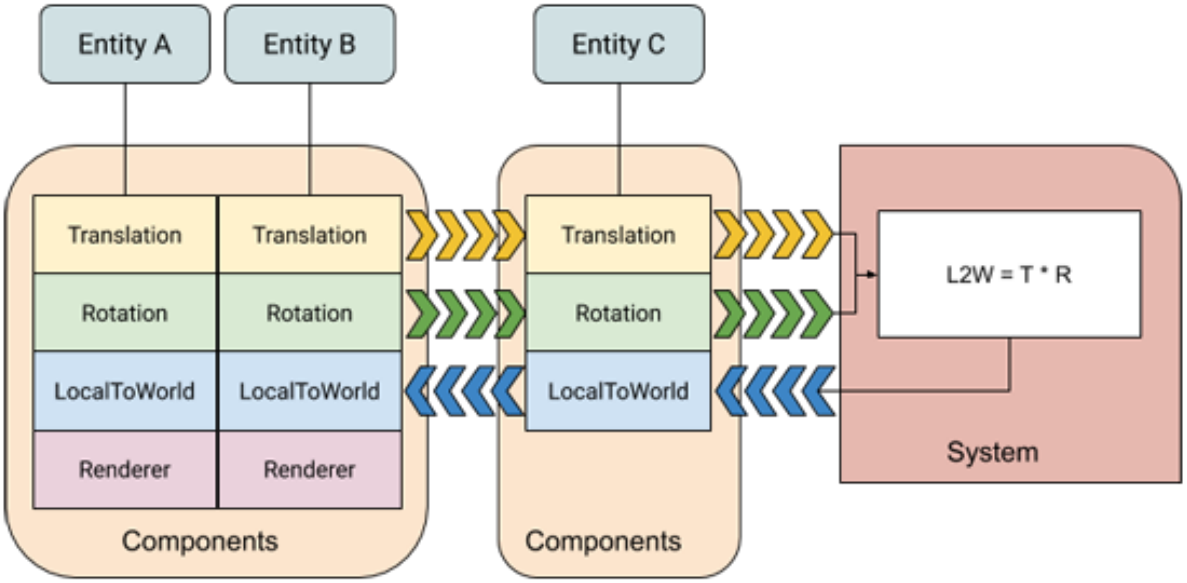
Kuva 15 Toteutustapojen havainnollistava kuva



4.7 Entity component system (ECS)

Entity component system (ECS) on videopelien kehityksessä käytetty malli. Entity component sytem suosii olio-ohjelmoinnissa laajasti käytetyn perinnän sijaan koostumukseen nojaavaa periaatetta, mikä luo pelikehitykseen joustavuutta sekä auttaa pelikehittäjiä tunnistamaan pelin eri entiteetit niihin liitettyjen komponenttien perusteella. Entiteetti edustaa yhtä objektia pelissä. Entiteetti itsessään on vain uniikki tunniste, minkä avulla pelin eri entiteetit voidaan erottaa toisistaan tai minkä avulla yksittäiseen entiteettiin voidaan kohdistaa toimenpiteitä. Entiteetin toiminnallisuudet määrittelevät sille annetut komponentit. Komponentit ovat tyypillisesti luokkia, jotka kapseloivat sisäänsä primitiivisiä datatyyppejä ilman toiminnallisuuksia. Systeemit käsittelevät entiteettejä, jotka sisältävät systeemille oleellisia komponentteja. (Terra, 2023)

Kuva 16 Entity component system havainnekuva. (Unity Technologies, n.d.a)



Kuvassa 16 entiteetteihin on asetettu komponentteja (translation, rotation) joita systeemi käyttää laskeakseen LocalToWorld komponentille arvon.

5 Suunnitelma

Opinnäytetyötä varten toteutettavan pelimoottorin on tarkoitus sisältää pelimoottoriarkkitehtuuri osuudessa mainitut pelimoottorin ydinkomponentit. Pelimoottori toteutetaan Monogame-ohjelmistokehyksellä. Peliä varten moottoriin toteutetaan myös entity component system -malli, minkä avulla hoidetaan pelin entiteettien käsittely sekä pelilogiikan toteutus.

Pelisilmukasta pyritään tekemään korkealla tasolla niin yksinkertainen, että pelimoottorin toimintalogiikka on helposti ymmärrettävissä. Ennen varsinaisen pelisilmukan käynnistämistä ladataan pelimoottorin käyttöön tulevat materiaalit, sekä alustetaan grafiikkaohjaimen hallintaan vaadittavat komponentit.

Materiaalien, kuten tekstuurien, ääniefektien sekä animaatioiden lataamiseen käytetään Monogame:n Content Pipelineä. Content Pipeline pystyy kääntämään pakkaamattoman sisällön kohdealustalle optimoituun muotoon. Content Pipeline ei ole kuitenkaan suunniteltu edellä mainittujen materiaalien metadatan käsittelyyn, joten Content Pipelinelle toiminnallisuutta on jatkettava metadatan käsittelyyn soveltuvaksi kokonaisuudeksi. Tiedostojen metadata voidaan toteuttaa usealla eri tavalla, esimerkiksi binäärimuodossa tai jonkinlaisena tekstitiedostona. Binäärimuotoinen data on huomattavasti nopeampaa käsitellä, sekä binääritiedostot ovat pienempiä kooltaan sillä ne eivät sisällä sovelluskehityksessä tyypillisesti käytettyjen tekstimuotoisten, kuten XML tai JSON, tiedostojen tapaan erotinmerkkejä tai datakenttien tunnisteita (Iliev, 2024). Opinnäytetyötä varten toteutettavan pelimoottorin käyttämien materiaalien metadata kirjoitetaan JSON-tiedostomuotoon sen helppolukuisuutensa ja vaivattomuutensa vuoksi, sillä pelimoottorilla toteutetun pelin suorituskykyvaatimukset ovat olemattomat, joten metadatan käsittelyyn kuluva aika ei ole suuressa roolissa.

Animointi toteutetaan sprite animointitekniikalla. Tämä animointitapa soveltuu erinomaisesti 2-uloitteisen pelin toteutukseen. Animaation kuvakehykset sisältämä kuvatiedosto on jokin yleisesti käytettävä tiedostotyyppi, kuten PNG tai JPEG. Kuvatiedosto sisältää animaation jokaisen kuvakehyksen peräkkäisinä kuvina. Animaation kuvatiedoston lisäksi animaatiotiedostojen käsittelyn automatisointiin luodaan JSON metadata tiedosto. Metadata tiedosto sisältää monenlaista tietoa animaation kuvatiedostosta, kuten tiedon kuvakehyksien määrästä, ja kuvakehyksen vaihtotaajuuden sekä kuvakehyksen koon.

Fysiikan mallinnus on tarkoitus toteuttaa mallintamalla painovoiman vaikutusta liikkuvien pelientiteettien osalta. Tasohyppelypelin tapaan pelaajan hahmon on kyettävä hyppäämään

eri suuntiin, hahmon on kyettävä seisomaan tasojen päällä ja törmäämään niihin sekä pelihahmon on kyettävä poimimaan esineitä pelitasoilta. Liikkuvien entiteettien törmäyksien tarkastelu toteutetaan läpimenon tarkasteluun nojaten.

Entity component system pyritään toteuttamaan tyypillisellä tavalla, missä entiteetti määritellään siihen liitettyjen komponenttien perusteella tyyppihierarkian sijaan. Järjestelmät päivittävät entiteettien komponenttien tietoja, mikäli entiteettiin on liitetty järjestelmään ohjelmoitujen funktioiden toimintaan vaadittavat komponentit.

Toteutettavan pelin on tarkoitus olla melko yksinkertainen loputon sivusuuntainen tasohyppelypeli, missä uusia tasoja generoidaan pelaajan edetessä pelinäkymässä eteenpäin tasolta toiselle. Pelin tavoite on yrittää hankkia pisteitä keräämällä jonkinlaisia esineitä, sekä tavoitella suurempaa tulosta kuin edellisillä kierroksilla. Pelissä on tarkoitus olla jonkinlainen valikkorakenne yksinkertaisilla toiminnoilla varustettuna. Pelaajan edistymistä pelissä ei ole tarkoitus tallentaa, vain edellisten yritysten huippupistemäärä on tarpeen säilöä pelaajan tietokoneella, että seuraavilla pelikerroilla voidaan esittää tavoitepistemäärä. Toteutettavan pelin visuaaliseen ilmeeseen ei panosteta, joten pelin käytössä olevat grafiikat sekä ääniefektit ladataan jostain avoimen lisenssin materiaaleja tarjoavasta palvelusta.

Peliä varten toteutetaan tasogeneraattori, mikä luo loputtomasti tasoja pelaajan edetessä pelissä eteenpäin. Tasoille asetetaan kolikoita, joita pelaajan on kerättävä. Tasogeneraattori sekä valikot toteutetaan irrallisena kokonaisuutena, joten ne eivät ole osa entity component system mallia.

6 Toteutus

Opinnäytetyötä varten käytetään Macbook Air M1 tietokonetta sekä Visual Studio Code -tekstieditoria. Monogame on alustariippumaton ohjelmistokehys, joten Unix spesifistä toteutusta ei pelimoottoriin tule. Pelimoottori on siis toteutettavissa myös esimerkiksi Windows käyttöjärjestelmällä. Pelimoottorin toteutuksen osuus alkaa oletuksella, että kehitysympäristöön on asennettu .NET sekä Monogame-ohjelmistokehykset.

Opinnäytetyötä varten toteutettavan pelin nimeksi annetaan *SimplePlatformer*. Kaikki pelin grafiikka sekä ääniefektit on ladattu opengameart.org sivustolta.

6.1 Pelisilmukan runko

Monogame-ohjelmistokehys tarjoaa muuttujia sekä ylikirjoitettavia metodeita pelisilmukan toteuttamista varten. Tätä toteutusta varten käytetään seuraavia Monogame:n ylikirjoitettavia metodeita:

- Initialize
    - Kutsutaan kerran pelin käynnistyessä, käyttötarkoitus on ei-graafisten materiaalien lataus
- LoadContent
    - Kutsutaan kerran, käyttötarkoitus on pelin graafisten materiaalien lataus
- Update
    - Kutsutaan jokaisella ruudunpäivityksellä, käyttötarkoitus on pelitilan laskenta
- Draw
    - Kutsutaan jokaisella ruudunpäivityksellä, käyttötarkoitus on grafiikan renderöinti

Pelin pääluokka perii Monogame-ohjelmistokehyksen *Game* luokan, mikä on pelin aloituspiste. Game luokka käsittelee peliruudun ja grafiikan päivittämistä sekä kutsuu Update ja Draw metodeita pelisilmukassa.

Kuva 17 Pelisilmukan ylikirjoitettavat metodit

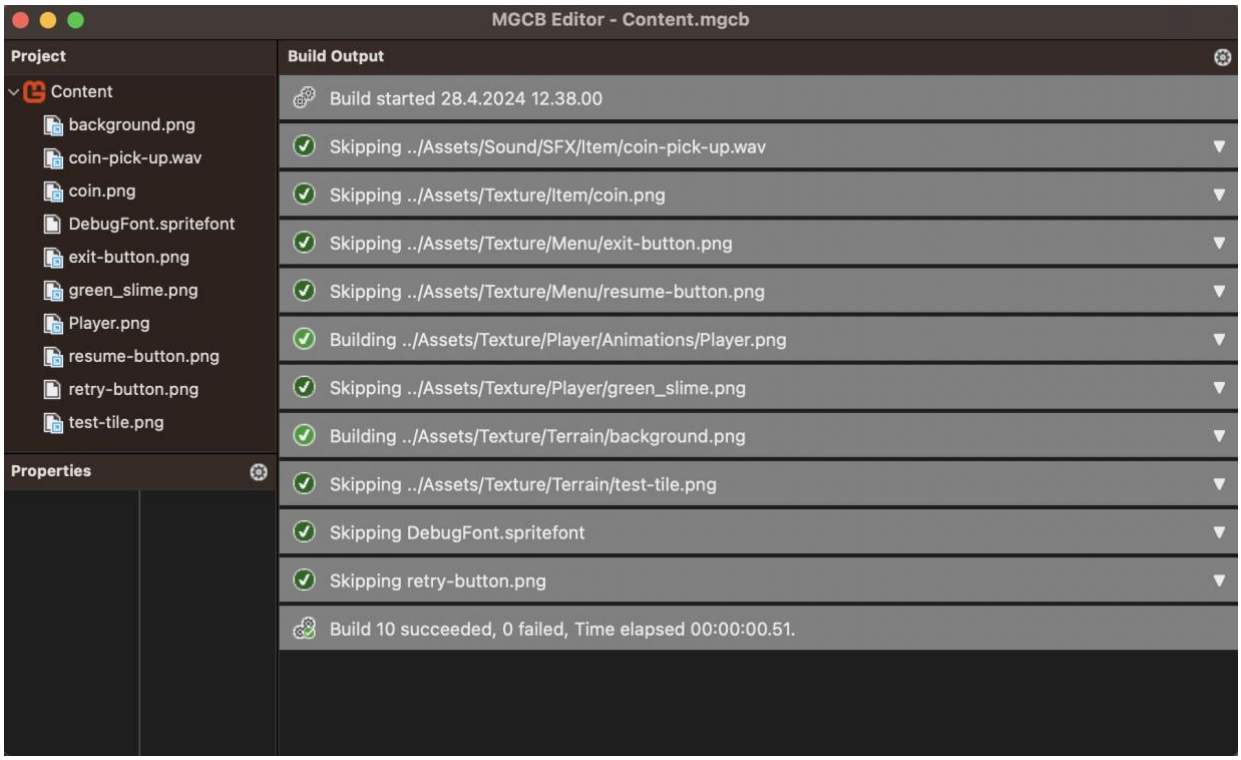
```
using Microsoft.Xna.Framework;

namespace GameProject
{
    2 references
    public class SimplePlatformer : Game
    {
        1 reference
        public SimplePlatformer()
        {
        }
        0 references
        protected override void Initialize()
        {
        }
        0 references
        protected override void LoadContent()
        {
        }
        0 references
        protected override void Update(GameTime gameTime)
        {
        }
        0 references
        protected override void Draw(GameTime gameTime)
        {
        }
    }
}
```

6.2 Pelimateriaalien lataus

Peliä varten tarvitaan erilaisia pelimateriaaleja, kuten sprite animaatioita, sprite kuvakehyksiä sekä ääniefektejä. Tarvittavat materiaalit käännetään optimoituun muotoon käyttäen Monogamen Content Pipeline työkalua. Ladatuista materiaaleista puuttuu kuitenkin metadata, joten ladattujen pelimateriaalien määrittely olisi kirjoitettava lähdekoodiin. Tämä ei ole optimaalinen ratkaisu, joten metadataa varten on luotava tiedostoformaatti sekä metadatan käsittelyjärjestelmä.

Pelimateriaalit on tallennettu pelin juureen Assets nimiseen kansioon. Seuraavaksi materiaalit käännetään Content Pipeline työkalulla Monogamelle optimaaliseen muotoon.

Kuva 18 Pelimateriaalin lataus Content Pipeline työkalulla



Tiedostojen metadata kirjoitetaan JSON-tiedostoon. Pelimateriaalista halutaan tietää ainakin seuraavat asiat:

- Nimi
- Materiaalin tyyppi
- Kohde-entiteetti
- Materiaalin polku

Nimi toimii materiaalin tunnisteena, jolla materiaalia voidaan hakea muiden saman tyyppisten materiaalien joukosta. Materiaalin tyyppi määrittelee, onko kyseessä esimerkiksi ääniefekti tai animaatio. Kohde-entiteetti määrittelee, onko kyseinen materiaali tarkoitettu jonkin tietyn entiteettityypin käyttöön (esimerkiksi pelaaja, esine tai taso). Materiaalin polku sisältää materiaalin fyysisen polun tietokoneella.

Metadatalle luodaan kantaluokka, joka määrittelee edellä mainitut tiedot pelin materiaalista.

Kuva 19 Asset kantaluokka

```
public class Asset
{
    public AssetType AssetType { get; init; }
    public EntityType EntityType { get; init;}
    public string Name { get; set;}
    public string AssetPath { get; set; }
    public string ContentName { get; set; }
}
```

Kantaluokasta periytetään johdetut luokat pelin animaatioille, sprite kehyksille, ja ääniefekteille, jotka määrittelevät näiden materiaalien metadataa.

6.2.1 Animaatio

Pelissä käytetään sprite pohjaista animointitekniikkaa, missä animaation jokainen kuvakehys on yhdessä tiedostossa. Animaatio pelimateriaalin metadatan rakenne sisältää seuraavat tiedot:

- Kuvakehyksien erottajan leveys pikseleinä
- Kuvakehyksen kesto millisekunneissa
- Kuvakehyksen leveys ja korkeus
- Animaatiotyyppi
- Kuvakehyksen skaalaus

Kuva 20 Asset kantaluokasta johdettu AnimationAsset luokka

```
public class AnimationAsset : Asset
{
  public int KeyFrameSeparatorWidth { get; set; } = 0;
  public List<Animation> Animations { get; set; }
  public float Scale { get; set; }
}
```

Animation luokka määritelee yksittäisen animaation kuvakehyksien tietoja, muun muassa kuvakehyksen leveys ja korkeus, kuvakehyksen kestoaika sekä kuvakehyksen suunnan, jos sellainen on. Esimerkiksi kävelyanimaatio on piirretty niin, että hahmon oletuskulkusuunta on oikealle, joten pelissä liikuttaessa vasemmalle on kuvakehys peilattava Y-akselin poikki. Yksittäinen *AnimationAsset* tyypin pelimateriaali voi siis sisältää useamman animaation.

Kuva 21 Kolme animaatiota yhdessä kuvatiedostossa.

Kuvassa 21 ruutupohjalla on alkuperäinen sprite animaation kuvatiedosto. Kuvatiedostoon on selkeyttämisen vuoksi lisätty vihreä nuoli indikoimaan kuvakehysten erottajan leveyttä, oranssit katkoviivat indikoimaan kuvakehyksen leveyttä ja korkeutta, sekä violetti laatikko indikoimaan kuvatiedostossa olevaa kolmea animaatiotyyppiä (Idle, Walk, Jump).

6.2.2 Sprite grafiikka

Animaatio pelimateriaalista poiketen sprite kuvakehysmateriaali voi sisältää useita kuvakehyksiä, mutta niitä ei ole tarkoitus toistaa sarjassa animaation tapaan, vaan kuvakehykset on tarkoitettu käytettävän yksittäin. Tämän tyyppisiä pelimateriaaleja voidaan käyttää esimerkiksi pelin valikoiden painikkeiden kuvakkeina tai tasohyppelypelin tasojen grafiikkana, tai muina graafisina elementteinä mitkä pysyvät muuttumattomina peliajan muuttuessa.

Kuva 22 Pelitason grafiikka

Sprite-kuvakehys materiaalista määritellään kuvakehyksen korkeus ja leveys, skaalaus sekä kuvakehyksen poikkeama. *Texture2D* tyypin muuttuja on referenssi ladattuun kuvatiedostoon.

Kuva 23 Asset kantaluokasta johdettu SpritesheetAsset luokka

```
public class SpritesheetAsset : Asset
{
    public int SpriteHeight { get; set; }
    public int SpriteWidth { get; set; }
    public int XOffSet { get; set;}
    public int YOffSet { get; set;}
    public float Scale { get; set; }
    public Size SpriteSize { get; set; }
    public Texture2D Texture { get; set; }
}
```

6.2.3 Ääniefektit

Ääniefektien metadata koostuu vain *Asset* kantaluokan jäsenistä. Varsinainen ääniefekti ladataan Monogamen *ContentManager* luokan toteutuksen avulla johdettuun *SoundAsset* luokkaan.

6.2.4 Lataus ja hallinta

Pelimateriaalien hallinnasta vastaa AssetManager luokka, jonka tehtävä on säilöä ladattuja pelimateriaaleja kokoelmissa, sekä tarjota materiaalien kuluttajille tarvittavat metodit oikean pelimateriaalin lataukseen.

Kuva 24 AssetManager luokka

```
public static class AssetManager
{
  private static AssetLoader AssetLoader;
  private static IEnumerable<Asset> Assets;
  private static ContentManager ContentManager;
  public static SpriteFont DebugFont { get; set; }

  static AssetManager()
  {
    AssetLoader = new AssetLoader();
  }

  public static void Init(ContentManager manager)
  {
    ContentManager = manager;
  }
  public static void LoadAssets()
  {
    Assets = AssetLoader.LoadAssetData(ContentManager);
    DebugFont = ContentManager.Load<SpriteFont>("DebugFont");
  }
  public static List<SpritesheetAsset> GetSpriteAsset(EntityType target) …
  public static T GetAsset<T>(EntityType entityTarget) where T : Asset …
  public static IReadOnlyCollection<Animation> GetAnimations(EntityType entityTarget, AnimationType animationType) =>…
  public static IReadOnlyCollection<Animation> GetAnimations(EntityType entityType, AnimationType animationType, string name) =>…
  public static IReadOnlyCollection<SoundAsset> GetSoundEffects(EntityType entityType) =>…
  public static IReadOnlyCollection<SoundAsset> GetSoundEffects(EntityType entityType, string name) =>…
}
```

Monogame tarjoaa *ContentManager* luokan Content Pipelinellä käännettyjen materiaalien lataukseen, mikä lisätään *AssetManager* luokan Init metodiin argumentiksi. *AssetManager* luokan Init metodia kutsutaan Monogamen pelisilmukan LoadContent metodissa, jolloin pelimoottori lataa muistiin saatavilla olevat pelimateriaalit. Pelimateriaalien metadatan lukemista varten *ContentManager* luokan toimintoja on laajennettava tukemaan myös JSON tiedostojen lukemista.

Pelimateriaalien lataamista ja metadatan lukemista varten luodaan *AssetLoader* luokka. *AssetLoader* lataa kaiken Assets-kansion sisällön, tulkitsee metadata-tiedoston perusteella pelimateriaalin käyttötarkoituksen sekä palauttaa listan ladatuista pelimateriaaleista *AssetManager*:lle.

Kuva 25 AssetLoader luokka

```
public class AssetLoader
{
    readonly string AssetPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Assets");
    private Dictionary<string[], Func<string, ContentManager, Asset>> AssetHandlers =
        new Dictionary<string[], Func<string, ContentManager, Asset>>();

    public AssetLoader()
    {
        AssetHandlers.Add(new string[] { ".png", ".jpeg", ".jpg", ".wav" }, LoadAssets);
    }

    public IEnumerable<Asset> LoadAssetData(ContentManager content)…

    private Asset LoadAssets(string filePath, ContentManager content)…
}
```

AssetLoader käy rekursiivisesti läpi kaikki Assets kansion alla olevat alihakemistot sekä niiden tiedostot, ja mikäli tiedostopääte on jokin *AssetHandlers* kokoelmaan määritellyistä tiedostopäätteistä, luetaan pelimateriaalin metadata-tiedosto *LoadAssets* metodilla.

Kuvassa 26 esitellään *LoadAssets* metodin animaatioiden lataukseen toteutettu koodilohko. JSON-tiedostomuodossa oleva metadata deserialisoidaan *Asset* tyyppin olioksi, jolloin metadatan perustella voidaan päätellä minkä tyyppinen pelimateriaali on kyseessä. Kun pelimateriaalityyppi on todettu, deserialisoidaan metadatatiedosto uudestaan kohdetyypin olioksi, jolloin pelimateriaalin metadata on luettu oikeassa muodossa.

Kuva 26 Animaatio pelimateriaalin käsittely

```
private Asset LoadAssets(string filePath, ContentManager content)
{
    var metadataFile = Path.ChangeExtension(filePath, ".json");
    var assetDef = JsonConvert.DeserializeObject<Asset>(File.ReadAllText(metadataFile));
    Asset ret = default;
    switch(assetDef.AssetType)
    {
        case AssetType.Sprite: …

        case AssetType.Animation:
        {
            var asset = JsonConvert.DeserializeObject<AnimationAsset>(File.ReadAllText(metadataFile));
            var texture = content.Load<Texture2D>(asset.ContentName);
            asset.Id = Guid.NewGuid().ToString();
            asset.AssetPath = filePath;
            var cumulativeHeight = 0;
            foreach(var animation in asset.Animations)
            {
                var keyFrameCumulativeDuration = 0;
                var keyFrames = animation.KeyFrames.OrderBy(k => k.Order).ToList();
                animation.Texture = texture;
                for(int i = 0; i < keyFrames.Count; i++)
                {
                    var keyframe = keyFrames[i];
                    keyframe.SpriteSource = new Microsoft.Xna.Framework.Rectangle
                    {
                        X = i * animation.SpriteWidth + (i * asset.KeyFrameSeparatorWidth),
                        Y = cumulativeHeight,
                        Width = animation.SpriteWidth + keyframe.xOffset,
                        Height = animation.SpriteHeight + keyframe.yOffset,
                    };
                    keyframe.Scale = asset.Scale;
                    keyframe.DurationFromStart = keyFrameCumulativeDuration;
                    keyFrameCumulativeDuration += keyframe.Duration;
                }
                cumulativeHeight += animation.SpriteHeight;
                animation.Duration = keyFrameCumulativeDuration;
            }
            ret = asset;
            break;
        }
        case AssetType.SoundEffect: …
    }
    System.Diagnostics.Debug.WriteLine($"{ret.AssetType} {Path.GetFileNameWithoutExtension(filePath)} added to asset collection.");
    return ret;
}
```

Kun kaikki Assets kansion pelimateriaalit on käsitelty, palautetaan lista ladatuista pelimateriaaleista *AssetManager:lle*.

6.3 Entity component system toteutus

Pelimaailman jokainen entiteetti on kokoelma komponentteja, pois lukien pelin valikot.

6.3.1 Entiteetti

Entiteetti sisältää uniikin tunnisteen, minkä avulla entiteetit ovat eroteltavissa toisistaan. Komponentit määrittelevät entiteetin toimintoja. Jokainen entiteetille asetettu komponentti lisätään entiteetin komponenttitaulukkoon, mistä komponentti on haettavissa sille määritellystä lohkosta.

Kuva 27 Entiteetti luokka

```
public class Entity
{
    public string ID { get; set; } = Guid.NewGuid().ToString();
    public EntityType EntityType { get; set; }
    public Component[] Components;
    private int AttachedComponents { get; set; }

    public Entity()
    {
        ID = Guid.NewGuid().ToString();
        Components = new Component[Globals.Constants.Components.MaxComponentCount];
    }

    public void AttachComponent(Component component)…

    public void AttachComponents(params Component[] components)…

    public void RemoveComponent(Component component) …

    public T GetComponent<T>() where T : Component…

    public bool HasComponent<T>() where T : Component…

    public void Render()…
}
```

Entiteettiin asetettujen komponenttien tilatietoa ylläpidetään *AttachedComponents* muuttujassa. Muuttuja on kokonaisluku, minkä jokainen bitti määrittelee, onko komponentti saatavilla komponenttitaulukon sille varatusta lohkosta. Aktiivinen komponentti entiteetissä tarkoittaa, että *AttachedComponents* muuttujassa komponentille määritellyn bitin arvo on 1.

Kuva 28 Komponentin asettaminen entiteetin muistilohkoon

```
public void AttachComponent(Component component)
{
    var index = ComponentRegister.GetComponentIndex(component);
    AttachedComponents = ComponentRegister.RegisterComponent(index, AttachedComponents);
    Components[index] = component;
}
```

Komponentin poistaminen entiteetistä tapahtuu asettamalla komponentille määritellyn bitin arvoksi 0 ja asettamalla *null* arvo komponentin muistilohkoon.

Kuva 29 Komponentin poistaminen entiteetistä

```
public void RemoveComponent(Component component)
{
    var index = ComponentRegister.GetComponentIndex(component);
    AttachedComponents = ComponentRegister.UnregisterComponent(index, AttachedComponents);
    Components[index] = null;
}
```

Entiteetti luokka sisältää myös metodit minkä avulla voi tarkastella onko entiteettiin liitetty haettavaa komponenttia, sekä entiteetin renderöintiin tarkoitettu metodi mistä lisää renderöinnin toteutusta käsittelevässä osiossa.

6.3.2 Komponentti

Entiteetteihin liitettävät komponentit ovat abstrakteja luokkia, mistä itsestään ei ole tarkoitus luoda instansseja, vaan komponenttiluokka määrittelee komponentille tyypillisimpiä piirteitä mitä laajennetaan johdetuissa luokissa. Komponetti luokka määrittelee komponentin tunnisteen, referenssin entiteettiin mihin komponentti on liitetty sekä tiedon onko komponentti aktiivinen.

Kuva 30 Abstrakti komponentti luokka

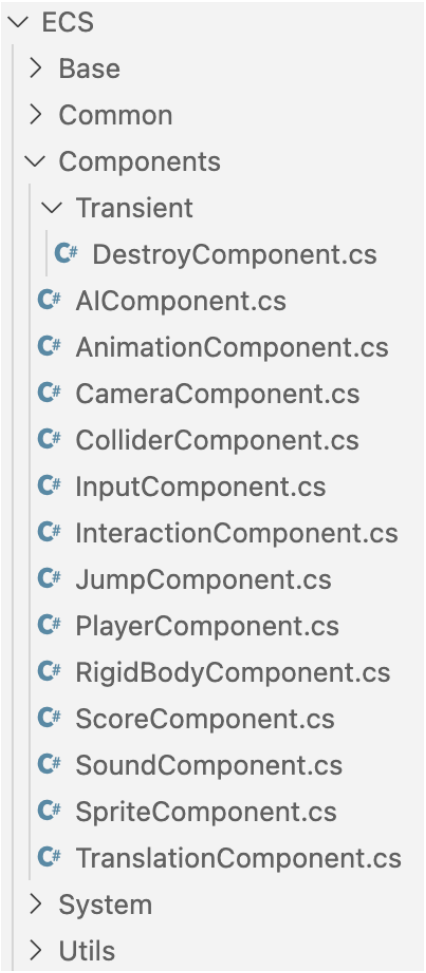
```
public abstract class Component
{
    public Entity Entity { get; init; }
    public string Id { get; } = Guid.NewGuid().ToString();
    public bool IsActive = true;
    public Component(Entity entity)
    {
        Entity = entity;
    }
}
```

Komponenttiluokka perimällä luodaan johdettuja luokkia mitä tullaan lisäämään entiteeteille kuvaamaan entiteettien toimintoja.

6.3.3 Johdetut komponentit

Kaikilla pelin johdetuilla komponenteilla on tärkeä rooli pelin toiminnassa, mutta oleellisimmat komponentit ovat syötekomponentti, törmäytinkomponentti, animaatiokomponentti, translaatiokomponentti, sekä sprite-kuvakehyskomponentti.

Kuva 31 Pelimoottorin johdetut komponentit



Jokaiselle johdetulle komponentille on olemassa vastaava systeemi, minkä tehtävä on käsitellä entiteettien komponentteja sekä päivittää komponenttien tilatietoja, mikäli entiteetillä on tarvittavat komponentit päivityksen suorittamiseen. Jokainen luotu johdettu komponentti lisätään sille vastaavan systeemin taulukkoon käsittelyä varten.

Kuvassa 32 esimerkkinä animaatiokomponentti. Komponentin rakentajametodissa komponentti rekisteröidään animaatiosysteemiin, jolloin kohdesysteemillä on referenssi luotuun instanssiin. Systeemi voi tarvittaessa iteroida kaikki sille rekisteröidyt komponentit läpi sekä tehdä tarvittavia toimenpiteitä komponenteille. Systeemien yksityiskohtaisemmasta toteutuksesta seuraavassa kappaleessa lisää.

Kuva 32 Animaatiokomponentti

```
public class AnimationComponent : Component
{
    public List<Animation> AvailableAnimations { get; set; }
    public Animation ActiveAnimation { get; set; }
    public DateTime ActivationTime { get; set; }
    public AnimationComponent(Entity entity) : base(entity)
    {
        AnimationSystem.RegisterComponent(this);
    }

    public bool HasAnimation(AnimationType animationType) …

    public void Init(AnimationType animationType) …

    public KeyFrame GetAnimationKeyFrame(double deltaTime) …
}
```

6.3.4 Systeemi

Komponenttien tapaan, systeemille on määritelty abstrakti kantaluokka, jonka jokaisen johdetun systeemin on perittävä. Kantaluokan toteutus on yksinkertainen; se sisältää geneerisen kokoelman komponentteja, sekä metodit komponenttien lisäämiseen ja poistamiseen kokoelmasta. Systeemit ovat toisistaan riippumattomia.

Kuva 33 Systeemin abstrakti kantaluokka

```
public abstract class BaseSystem<T> where T : Component
{
    protected static List<T> Components = new List<T>();
    public List<T> ActiveComponents
        { get => Components.Where(c => c.IsActive).ToList(); }

    public static void RegisterComponent(T component)
    {
        Components.Add(component);
    }

    public static void UnRegisterComponent(T component)
    {
        var c = Components.SingleOrDefault(c => c.Id == component.Id);
        if(c != null)
        {
            Components.Remove(c);
        }
    }
}
```

Johdetut systeemiluokat määrittelevät pelilogiikan. Systeemit eivät ole rajoitettu käsittelemään vain systeemille suunnattuja komponentteja, vaan ne voivat referoida käsiteltävän komponentin entiteettiä sekä hakea entiteetin muitakin komponentteja.

Kuvassa 34 systeemin kantaluokasta johdettu luokka. Luokan metodin tarkoitus on päivittää kaikkien *RigidBodyComponent*-komponentin sisältävien entiteettien sijainti näytöllä ruudunpäivityksen aikana. Sijainnin päivittämiseen tarvitaan myös *TranslationComponent*-komponentti, mikä sisältää ruudunpäivityksen sijainnin muutosvektorin.

Kuva 34 Entiteetin position päivitys

```
public class RigidBodySystem : BaseSystem<RigidBodyComponent>
{
    public void UpdatePositions() {
        foreach(var component in ActiveComponents)
        {
            var entity = component.Entity;

            var translation = entity.GetComponent<TranslationComponent>();
            var rigidBody = entity.GetComponent<RigidBodyComponent>();

            if(translation.IsStatic || !translation.IsActive) continue;

            rigidBody.EntityPosition = new Vector2(
                rigidBody.EntityPosition.X + translation.Velocity.X,
                rigidBody.EntityPosition.Y + translation.Velocity.Y);

            rigidBody.Bounds = new BoundingBox(
                min: new Vector3(
                    rigidBody.Bounds.Min.X + translation.Velocity.X,
                    rigidBody.Bounds.Min.Y + translation.Velocity.Y, 0),
                max: new Vector3(
                    rigidBody.Bounds.Max.X + translation.Velocity.X,
                    rigidBody.Bounds.Max.Y + translation.Velocity.Y, 0)
            );
            Debug.AddMessage("Velocity on frame end",
                $"X:{translation.Velocity.X.ToString("0.00")}",
                $"Y:{translation.Velocity.Y.ToString("0.00")}");
        }
    }
}
```

6.4 Rakentajaluokat

Pelimaailman hahmot, esineet ja tasot on yksinkertaisinta luoda keskitetysti rakentajaluokissa. Kuvassa 35 pelaajahahmon luontiin toteutettu rakentajaluokka. Pelaajahahmolle asetetaan sen tarpeisiin vaadittavat komponentit sekä komponenttien tilatiedoille annetaan jotkin lähtöarvot.

Kuva 35 Pelaajahahmon rakentajaluokka

```
public static class PlayerFactory
{
  public static Entity GeneratePlayer()
  {
    var player = new Entity { EntityType = EntityType.Player };
    var startingPosition = new Vector2(0, -150);

    player.AttachComponents(
        new SpriteComponent(player) { },
        new CameraComponent(player)…
        new AnimationComponent(player)…
        new PlayerComponent(player)…
        new InputComponent(player)…
        new TranslationComponent(player)…
        new ColliderComponent(player)…
        new JumpComponent(player)…
        new ScoreComponent(player)
      );

    player.GetComponent<AnimationComponent>().Init(AnimationType.Idle);
    var sprite = player.GetComponent<SpriteComponent>();

    player.AttachComponent(new RigidBodyComponent(player)
    {
      EntityPosition = startingPosition,
      Bounds = new BoundingBox(
        min: new Vector3(startingPosition.X, startingPosition.Y, 0),
        max: new Vector3(
          startingPosition.X + sprite.SpriteSize.Width * sprite.Scale,
          startingPosition.Y + sprite.SpriteSize.Height * sprite.Scale, 0))
    });

    return player;
  }
}
```

6.5 Tasojen luonti

Tasohyppelypelin tasoja ylläpitää ja luo *MapManager* luokka. *MapManager* tarkastelee pelaajan sijaintia ruudulla, sekä käynnistää tasojen luontiprosesseja, mikäli pelaaja lähestyy näytön oikeaa reunaa. *MapManager* käyttää erillistä karttageneraattori luokkaa pelitasojen luontiin. Karttageneraattori luo sille annettujen määrittelyjen perusteella sprite grafiikkaan pohjautuvia tasoja, jotka sisältävät tarvittavat komponentit, että pelaajan törmäytinkomponentti osaa havaita törmäyksen tason törmäytinkomponentin kanssa.

Kuva 36 Karttageneraattorin toteutus

```
public class MapGenerator
{
    public List<Entity> GeneratePlatform(PlatformProps props)
    {
        var tiles = new List<Entity>();
        var sprite = AssetManager.GetSpriteAsset(EntityType.Terrain).First();
        var tilePosition = props.StartingPosition;

        for(int i = 0; i < props.TileCount; i++)
        {
            tiles.Add(GenerateTile(tilePosition, sprite));
            tilePosition = new Vector2(tilePosition.X + sprite.SpriteWidth, tilePosition.Y);
        }
        return tiles;
    }

    private Entity GenerateTile(Vector2 position, SpritesheetAsset asset)
    {
        var tile = new Entity { EntityType = EntityType.Terrain };
        tile.AttachComponents(
            new SpriteComponent(tile) {…
            new RigidBodyComponent(tile){…
            new TranslationComponent(tile) { IsStatic = true },
            new InteractionComponent(tile) { Pickable = false, Unmovable = true },
            new ColliderComponent(tile)
        );
        return tile;
    }
}
```

6.6 Pelivalikot

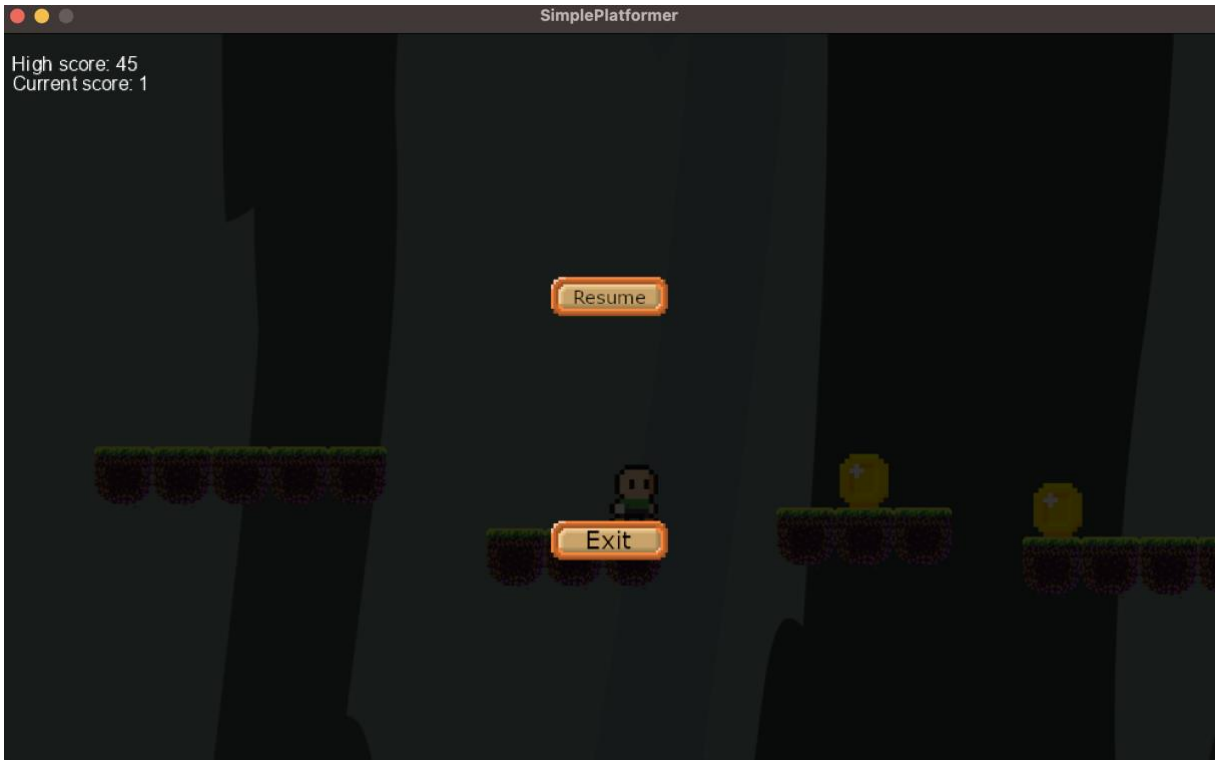
Pelivalikot toteutettiin irrallisena kokonaisuutena olemassa olevasta ECS-järjestelmästä. Pelivalikon painikkeet koostuvat sprite kehyksestä, sekä Action delegaatista mihin määritellään haluttu toiminto, kun pelaaja painaa pelivalikon painiketta hiirellä. Pelaajan *InputComponent* komponentti sisältää pelaajan antamien syötteiden tiedot, mikä sisältää myös hiirien painikkeiden painallukset. Kun pelaaja painaa ESC painiketta, tai kun pelihahmo putoaa tasolta vapaapudotukseen, valikko renderöidään tarvittavilla painikkeilla.

Kuva 37 Pelivalikko luokan rakentaja

```
static MenuManager()
{
    InitOverlay();
    var verticalMiddle = Globals.GraphicsDevice.PresentationParameters.BackBufferWidth / 2;
    MenuScreens.Add(new MenuScreen
    {
        MenuType = MenuType.IngameMenu,
        MenuItems = new List<MenuItem>()
        {
            new MenuItem( …
            new MenuItem(
                texture: ExitButtonAsset.Texture,
                position: new Vector2(verticalMiddle - ResumeButtonAsset.SpriteWidth / 2, ButtonSpacing * 2),
                onClick: () => Globals.ExitInitiated = true)
        }
    });
    MenuScreens.Add(new MenuScreen
    {
        MenuType = MenuType.GameLostMenu,
        MenuItems = new List<MenuItem>()
        {
            new MenuItem( …
            new MenuItem( …
        }
    });
}
```

Pelivalikko koostuu läpinäkyvästä tummasta kerroksesta, mikä peittää käynnissä olevan pelin, sekä tumman kerroksen päälle piirrettävistä painikkeista. Itse pelistä poiketen, pelivalikoiden toiminnot ovat aktivoitavissa hiiren painikkeilla.

Kuva 38 Yksinkertainen pelivalikko



6.7 Pelin toteutus

Pelin toteuttaminen aloitetaan lisäämällä *Initialize* metodiin MonoGamen *ContentManagerin* konfigurointi sekä pelimoottoriinstanssin luonti. Peli myös lukitaan toimimaan 60 ruudunpäivityksellä sekunnissa ja hiiren kursori jätetään näkyviin. *Initialize* metodia kutsutaan kerran pelin käynnistyessä.

Kuva 39 Initialize metodi

```
protected override void Initialize()
{
    Content.RootDirectory = "Content";
    gameEngine = new GameEngine();
    IsFixedTimeStep = true;
    IsMouseVisible = true;
    base.Initialize();
}
```

Seuraavaksi ladataan pelin kaikki materiaalit metadatan perusteella sekä initialisoidaan pelimoottori luomalla uudet instanssit jokaisesta ECS-järjestelmän systeemistä. Pelin huippupistemäärää sekä pelaajan reaaliaikaisen pistemäärän visualisointia varten luodaan

ScreenWriter luokka, minkä tehtävä on renderöidä tekstiä näytölle. *AssetManager* luokan avulla peliin ladataan yksinkertainen fontti (kts. Kuva 24 AssetManager luokka) *ScreenWriter* luokan käyttöön. *ScreenWriter* luokalla voidaan myös piirtää kehitysaikaista tekstiä näytölle.

Kuva 40 Pelimateriaalien lataus ja pelin initialisointi

```
protected override void LoadContent()
{
    gameEngine.Init(Content);
    gameEngine.BuildGame();
}
```

```
public void Init(ContentManager manager)
{
    AssetManager.Init(manager);
    AssetManager.LoadAssets();
    ScreenWriter = new ScreenWriter(AssetManager.DebugFont);
}
public void BuildGame()
{
    Entities = new Dictionary<string, Entity>();
    MapManager = new MapManager();
    InputSystem = new InputSystem();
    ColliderSystem = new ColliderSystem();
    TranslationSystem = new TranslationSystem();
    RigidBodySystem = new RigidBodySystem();
    AnimationSystem = new AnimationSystem();
    CameraSystem = new CameraSystem();
    DestroySystem = new DestroySystem();
    SoundSystem = new SoundSystem();
    ScoreSystem = new ScoreSystem();

    MapManager.CreateInitialMap();
    Player = PlayerFactory.GeneratePlayer();
    Entities.Add(Player.ID, Player);

    MenuManager.CloseMenu();
}
```

Tässä vaiheessa kaikki tarvittavat valmistelut on tehty pelin toteutusta varten. Pelimoottorin *Init* metodilla initialisoidaan pelimateriaalien latausjärjestelmä sekä ladataan kaikki pelin käyttämät materiaalit. Pelimoottorin *BuildGame* metodin alustaa kaikki ECS-järjestelmän systeemit, luo ensimmäisen pelitason mikä tullaan renderöimään myöhemmässä vaiheessa sekä luo pelaajahahmon sille tarkoitetulla rakentajaluokalla. Lopuksi suljetaan avoimet pelivalikot, mikäli sellainen on auki.

6.7.1 Pelisilmukan toteutus

Varsinainen pelisilmukka toteutetaan Monogamen *Update* metodiin. Metodia kutsutaan jokaisessa ruudunpäivityksessä. Monogamen pelisilmukassa tarkastellaan pelin tilaa, esimerkiksi tarvitseeko peli käynnistää uudelleen, mikäli pelaaja haluaa hävitessään yrittää peliä uudelleen, tai pelin jos pelaaja painaa pelivalikon Exit-painiketta. Metodiin syötetään Monogamen toimesta *GameTime* tyyppinen olio, mikä sisältää tiedot ruudunpäivityksen kestosta sekä pelin kokonaissuoritusajasta. Nämä tiedot lisätään globaaleiksi muuttujiksi staattiseen *Globals* luokkaan, että ne ovat käytettävissä mistä tahansa projektissa. Tämän

jälkeen pelimoottorin pelisilmukka suoritetaan kutsumalla pelimoottorin *Update* metodia. *Update* metodin aikana pelimoottorin on tarkoitus laskea ruudunpäivityksen aikana tapahtuva muutos pelissä.

Kuva 41 Monogamen pelisilmukka

```
protected override void Update(GameTime gameTime)
{
    if(Globals.RetryInitated)
    {
        gameEngine.BuildGame();
        Globals.RetryInitated = false;
    }

    if(Globals.ExitInitiated)
    {
        Exit();
    }

    Globals.DeltaTime = gameTime.ElapsedGameTime.TotalMilliseconds;
    Globals.TotalGameTime = gameTime.TotalGameTime.TotalMilliseconds;

    gameEngine.Update();
    base.Update(gameTime);
}
```

Pelimoottorin pelisilmukka sisältää seuraavat vaiheet:

1. Pelaajan syötteen lukeminen
2. Syötteen perusteella toiminnon valinta
3. Tarkastus onko pelaaja pudonnut tasolta, häviö
4. Entiteettien tilasiirtymän laskenta ruudunpäivityksen aikana
5. Entiteettien törmäysten tarkastelu
6. Animaatiotilan ylläpito
7. Pelaajan seuraaminen ruudulla
8. Ääniefektien toistaminen
9. Tuhottavien entiteettien poistaminen

Kuva 42 Pelimoottorin pelisilmukka

```
public void Update()
{
    InputSystem.ReadInput();
    if(MenuManager.MenuOpen)
    {
        MenuManager.TrackInput(Player);
        return;
    }
    CheckGameLostCondition();
    MapManager.CheckMapState(CameraSystem.GetActiveCamera());
    TranslationSystem.CalculateFrameTranslation();
    DetectCollisions();
    RigidBodySystem.UpdatePositions();
    AnimationSystem.UpdateAnimationState();
    CameraSystem.Track(Player);
    SoundSystem.PlaySoundEffects();
    DestroySystem.CleanDestroyable(Entities);
}
```

Käyttäjän syöte luetaan käyttäen Monogamen *Keyboard.GetState* metodia, mikä palauttaa ruudunpäivityksen aikana painetut painikkeet *Keys* enumeraatio tyypin listana. Käyttäjän painamien painikkeiden tila tallennetaan *InputComponent* komponenttiin, ja riippuen painetuista painikkeista näytölle joko avataan pelivalikko, tai painettujen painikkeiden tilatietoa käytetään laskiessa ruudunpäivityksen aikana tapahtuvaa muutosta pelissä.

Pelin häviäminen tapahtuu putoamalla tasolta alas siten että pelaajan paluu tasolle ei ole enää mahdollista. Toisin sanoen, peli päättyy pelaajan ollessa alempana kuin pelissä matalimmalla oleva taso. Jokaisen pelissä olevan tason joukosta haetaan matalimmalla sijaitseva taso, ja pelaajan sijaintia verrataan tämän tason sijaintiin. Jos pelaaja on 500 pikseliä alinta tasoa alempana, ei pelaajalla varmasti ole mahdollisuutta palata takaisin millekään tasolle.

Kuva 43 Putoaminen pelitasolta

Pelissä häviöön asti kerätyt pisteet tallennetaan tekstitiedostoon tietokoneelle, sekä pelaajalle näytetään uudelleenyrityksen tai pelin sulkemisen pelivalikko. Jokaisen pelitason iterointi jokaisella ruudunpäivityksellä on laskennallisesti kallis operaatio, joten häviön laskenta rajoitetaan tapahtuvan kerran sekunnissa.

Kuva 44 Häviön tarkastelu

```
public void CheckGameLostCondition()
{
    if (Globals.TotalGameTime - LastGameLostChecked > GameLostCheckInterval)
    {
        var rigidBody = Player.GetComponent<RigidBodyComponent>();
        var lowestTile = MapManager.LowestTile();
        if(rigidBody.EntityPosition.Y - 500 > lowestTile.GetComponent<RigidBodyComponent>().EntityPosition.Y)
        {
            ScoreSystem.PersistScore();
            MenuManager.OpenMenuScreen(MenuType.GameLostMenu);
        }
        LastGameLostChecked = Globals.TotalGameTime;
    }
}
```

Seuraavaksi pelisilmukassa *MapManager* tarkistaa onko tarpeen generoida uusia tasoja laskemalla erotuksen näytön kuva-alan reunojen sekä kameran X-akselin ruudunpäivityshetken position perusteella. Pelimoottori generoi tasoja vain tarpeeseen.

Kuva 45 Generoitavien tasojen tarpeen laskenta

```
public void CheckMapState(CameraComponent camera)
{
    var mapRightmostTile = CurrentMap.Entities
        .Where(e => e.EntityType == EntityType.Terrain)
        .OrderByDescending(t => t.GetComponent<RigidBodyComponent>()?.EntityPosition.X).First();
    var viewportMiddle = Globals.GraphicsDevice.Viewport.Width / 2;
    var maxDistanceRight = camera.Position.X + viewportMiddle + MaxPlatformRenderDistance;

    if (mapRightmostTile.GetComponent<RigidBodyComponent>().EntityPosition.X <= maxDistanceRight)
    {
        #region GeneratePlatform…
    }
}
```

Pelisilmukan kolme seuraavaa metodikutsua (*CalculateFrameTranslation, DetectCollisions, UpdatePositions)* liittyvät pelaajahahmon liikkeeseen sekä törmäysten tarkasteluun.

CalculateFrameTranslation metodi tehtävä on laskea ennuste siitä, mihin entiteetit tulevat siirtymään tämän ruudunpäivityksen aikana. Entiteettien siirtymän laskentaan vaikuttavat muun muassa pelissä vallitseva painovoima, luettu syöte ja entiteetin sijainti (tasolla, ilmassa). Laskettu ennuste siirtymästä tallennetaan entiteetin *TranslationComponent* komponentin tietoihin seuraavien laskentavaiheiden käytettäväksi.

DetectCollisions nimensä mukaan tarkastaa mitkä pelin entiteeteistä tulevat törmäämään tämän ruudunpäivityksen aikana. Entiteettien välinen törmäys tunnistetaan laskemalla kahden liikkuvan entiteetin läpimenon leikkaus (kts. Kuva 13 Peliobjektin läpimenon tarkastelu), sekä reagoimalla todettuun päällekkäisyyteen muuttamalla entiteetin laskennallista liikerataa.

UpdatePositions metodi päivittää entiteetin fyysisen sijainnin tiedon sisältävää *RigidBodyComponent* komponenttia edellisten vaiheiden lopputuloksen perusteella.

Animaatiosysteemi muuttaa entiteettien aktiivisesti toistettavaa animaatiota riippuen vallitsevista olosuhteista. Esimerkiksi pelaajan ollessa liikkeessä aktivoidaan kävelyanimaatio, levossa aktivoidaan joutokäyntianimaatio ja niin edelleen. Pelissä käytettävät animaatiot ovat toteutettu niin että ne ovat loputtomiin toistettavissa sulavasti.

6.7.2 Grafiikan renderöinti

Grafiikan renderöinti alkaa Monogamen *Draw* metodissa. Metodissa asetetaan taustan väriksi tasainen musta sekä kutsutaan pelimoottorin grafiikan renderöintiin kehitettyä metodia.

Kuva 46 Monogame Draw metodi

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);
    gameEngine.RenderEntities();
    base.Draw(gameTime);
}
```

Renderöinti Monogame-ohjelmistokehyksellä aloitetaan aina kutsumalla *SpriteBatch* luokan *Begin* metodia, sekä renderöinti lopetetaan kutsumalla *SpriteBatch* luokan *End* metodi. *Begin* metodille annetaan argumentiksi *SpriteSortMode* tyypin enumeraatio, joka määrittelee missä järjestyksessä syötettyjen kuvien piirtäminen tapahtuu, sekä *BlendState* tyypin olio, joka määrittelee päällekkäin aseteltujen pikseleiden piirtotavan. Viimeinen argumentti on matriisi mikä sisältää tiedon renderöitävästä pelimaailman alueesta.

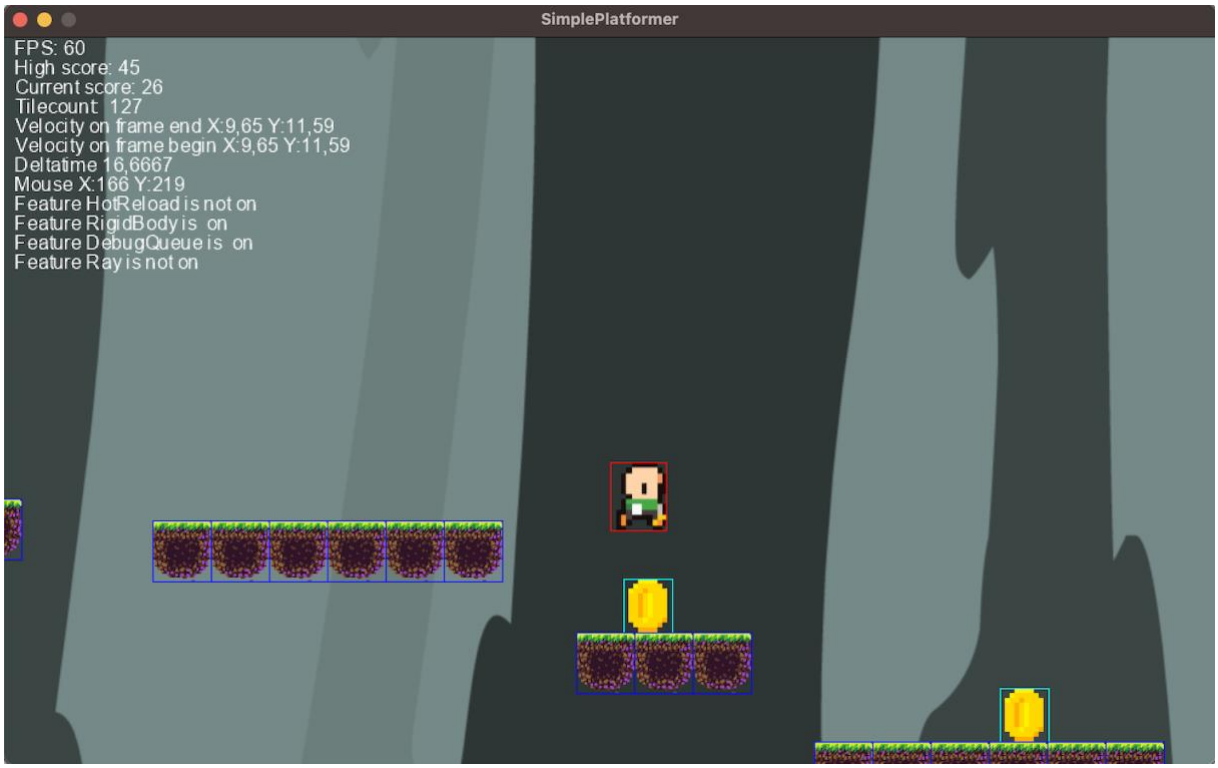
Entity luokkaan on toteutettu *Render* metodi (kts. Kuva 27 Entiteetti luokka), minkä avulla yksittäisen entiteetin grafiikka voidaan renderöidä näytölle, mikäli entiteettiin on liitetty renderöitävän kuvakehyksen sisältämä aktiivinen *SpriteComponent* komponentti. Renderöitävän entiteetin sijainti pelimaailmassa luetaan *RigidBodyComponent* komponentin tiedoista.

Kuva 47 Pelimoottorin renderöintimetodi

```
internal void RenderEntities()
{
    var camera = CameraSystem.GetActiveCamera();

    Globals.SpriteBatch.Begin(SpriteSortMode.FrontToBack, BlendState.AlphaBlend,
        null, null, null, null, camera.TranslationMatrix);

    RenderBackground();

    foreach(var id in Entities.Keys)
    {
        var entity = Entities[id];
        entity.Render();
    }

    Globals.SpriteBatch.End();

    if(MenuManager.MenuOpen)
    {
        Globals.SpriteBatch.Begin();
        MenuManager.Render();
        Globals.SpriteBatch.End();
    }

    Globals.SpriteBatch.Begin();
    var allTimeHighScore = ScoreSystem.GetHighScore();
    var currentPlayerScore = ScoreSystem.GetCurrentPlayerScore();
    ScreenWriter.DrawText($"High score: {allTimeHighScore.Points}");
    ScreenWriter.DrawText($"Current score: {currentPlayerScore}");
    Globals.SpriteBatch.End();

    #region RUNTIMEDEBUG…
}
```

Ohjelmakoodin alue *RUNTIMEDEBUG* sisältää debug moodissa käännetyn pelin suoritusaikaisen datan visualisointimenetelmiä, kuten kaikkien pelin entiteettien törmäytinobjektien visualisointi, kuvakehyksien päivitystaajuuden laskenta sekä hahmon laskennallisen nopeuden tilat pelisilmukan eri vaiheissa.

Kuva 48 Pelin suoritus debug moodissa



7 Pohdinta

Videopelien toteuttamiseen käytetään useasti valmista pelimoottoria. Yksinkertaisten pelien suunnittelu ja toteuttaminen oli itselleni ennestään tuttua Unity-pelimoottorilla, mutta pelien toteuttamiseen tarkoitettua ohjelmistokehystä en ennen tätä opinnäytetyötä ollut käyttänyt. Valmiit pelimoottorit piilottavat huomattavan määrän monimutkaisuutta rajapintansa alle, syvällisemmän tietotaidon hankkimisen yksi keino on rakentaa pelimoottori. Pelimoottorin rakentaminen on resurssi intensiivinen prosessi, joten pelimoottorin toteuttaminen ei usein ole taloudellisesti kannattavaa, etenkään Indie-pelistudiossa. Unity ja sekä muut vastaavat suositut pelimoottorit tarjoavat huomattavan määrän valmiita komponentteja peliprojektin toteuttamiseen. Useat pelimoottorit täydentävät omaa tarjontaansa jonkinlaisella pelimateriaalialustalla (esimerkiksi Unity Asset Store, Unreal Engine Marketplace), joka sisältää maksullisia sekä maksuttomia laajennuksia pelimoottoriin. Pelejä voidaan myös toteuttaa visuaalisella skriptityökalulla usealla pelimoottorilla, mikä mahdollistaa pelien kehittämisen, vaikka pelikehittäjä ei osaisi ohjelmoida pelimoottorille suunnatulla ohjelmointikielellä.

Pelimoottoriarkkitehtuurista on tarjolla suhteellisen laajasti kirjallisuutta sekä nettiartikkeleita. Pelimoottoreihin liittyvää tutkimustietoa ei ole niin laajasti saatavilla, ja saatavilla oleva tutkimustieto usein ei suoranaisesti liity pelimoottorin rakentamiseen. Pelien toteuttamiseen suunnattujen ohjelmistokehyksien käyttöaste on paljon pienempi verrattuna pelimoottoreihin, mikä tarkoittaa vielä niukempaa tutkimusmateriaalin ja nettiartikkeleiden määrää.

Pelimoottorin onnistunut toteuttaminen vaatii kehittäjältä paljon tietoa niin pelimoottoriarkkitehtuurista kuin ohjelmoinnista. Yksinkertaisenkin pelimoottori sisältää paljon liikkuvia osia, joiden ymmärtäminen vaatii paljon asiaan perehtymistä. Monogame pyrkii poistamaan osan tästä haasteesta tarjoamalla yksinkertaisen rajapinnan pelimateriaalien käsittelyyn, pelisilmukan toteuttamiseen, renderöintiin sekä näyttöohjainten hallintaan. Monogame itsessään ei pakota käyttäjää toteuttamaan peliänsä millään tietyllä tavalla, vaan se tarjoaa vankan raamin minkä avulla kehittäjä voi valita toteutustapansa itse. Monogamella ei ole suosittujen pelimoottorien tapaan suurta käyttäjäkuntaa, joten käyttöesimerkkien löytäminen on haastavaa. Monogamen dokumentaatiossa on parantamisen varaa koska käyttöesimerkkejä ei suuressa osassa dokumentaatiota ole lainkaan, eikä kuvaukset ole kovin kattavia.

Opinnäytetyö osoittaa, että 2D-pelimoottorin toteuttamiseen Monogame soveltuu hyvin. Monogamen tarjoama yksinkertainen rajapinta pelimoottorin perusominaisuuksien hallintaan auttaa kehittäjää keskittymään pelimoottoriarkkitehtuurin toteuttamiseen. Monogamen

Content Pipeline työkalu helpottaa pelimateriaalien lataamisessa sekä käsittelyssä, vaikkakin pelimateriaalien metadatan käsittelyä se ei suoranaisesti tue. Opinnäytetyössä käytetty JSON pohjainen pelimateriaalien metadata osoittautui hyväksi tavaksi ladata pelimateriaalien metadataa pelimateriaalityypin perusteella, mikä virtaviivaistaa pelimateriaalien käsittelyä.

Toteutetun pelimoottorin suorituskyvyn mittausta ei tehty rajallisen ajan vuoksi, sekä suorituskykyerojen testaus olisi melko haastavaa toteuttaa koska verrattavia pelimoottoreita on useita, sekä tasavertaisten testitapauksien luonti vaatisi paljon aikaa.

Opinnäytetyö osoittaa myös, että yksinkertaisen 2D-pelimoottorin toteuttaminen ei kuluta huomattavissa määrin kehitysaikaa, sekä ottaen huomioon esimerkiksi Unityn yritys muuttaa lisensointikäytäntöjään radikaalisti, voi lisenssivapaan avoimen lähdekoodin ohjelmistokehyksen käyttö peliprojektissa turvallisempaa myös tulevaisuutta ajatellen.

Lähteet

Ahamed, S. (2023) *Unity vs Unreal vs Godot - Comparison, Pros, Cons* https://imetatech.io/blog/unity-unreal-godot-comparison

Bradfield, C. (2018). *Godot Engine Development* Packt Publishing

Cranberg, Carl. (2009). *Character animation with Direct3D* Cengage Learning

Dalton, O. (2018). *Tall base* https://opengameart.org/content/tall-base

Darling, R. (2023). *12 popular games made with the Godot engine* https://www.thegamer.com/godot-engine-popular-games-best/

Dealessandri, M. (2020a). *What is the best game engine: is Monogame right for you?* https://www.gamesindustry.biz/what-is-the-best-game-engine-is-monogame-the-right-game-engine-for-you

Dealessandri, M. (2020b). *What is the best game engine: is Unity right for you?* https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you

Drake, J. (2023a). *24 Great Games That Use The Unity Game Engine* https://www.thegamer.com/unity-game-engine-great-games/

Drake, J. (2023b) *24 Great Games That Use The Unreal 4 Game Engine* https://www.thegamer.com/great-games-use-unreal-4-game-engine/

Dragonfly. (n.d). *Question: What is the difference between a game engine and a game library?* https://www.dragonflydb.io/faq/game-engine-vs-game-library

Epic Games. (2024) *Unreal Engine 5.3 Documentation* https://docs.unrealengine.com/5.3/en-US/

Erolin, J. (n.d). *What is Unreal Engine?* https://www.bairesdev.com/blog/what-is-unreal-engine/

Eventyr. (2023). *Game Development With Unreal Engine: Pros and Cons* https://eventyr.pro/blog/game-development-with-unreal-engine-pros-and-cons/

Foltz, B. (2011). *The Game Loop and Frame Rate Management* https://www.brandonfoltz.com/downloads/tutorials/The_Game_Loop_and_Frame_Rate_Management.pdf

Game Ace Studio. (2023). *The Best Game Engines: Research from Game-Ace Specialists* https://game-ace.com/blog/best-game-engines/

GameFromScratch. (2015). *GameDev Glossary: Library Vs Framework Vs Engine* https://gamefromscratch.com/gamedev-glossary-library-vs-framework-vs-engine/

Gregory, J. (2014). *Game engine architecture 2nd edition*

CRC Press

Gupta, A. (2023). *Tips for selecting the best game engine for your project* https://www.searchmyexpert.com/resources/game-development/choosing-game-engine

Godot. (n.d). *Godot Docs* https://docs.godot.community/getting_started/introduction/introduction_to_godot.html

Iliev, H. (2024). *JSON vs binary serialization* https://thatonegamedev.com/cpp/json-vs-binary-serialization/

Klappenbach, M. (2021) *What is a platformer game?* https://www.lifewire.com/what-is-a-platform-game-812371

Lewis, M & Jacobson, J. (2002). Game engines in scientific research *Communications of the ACM vol.45, 31*

Loach, R. (2023). *Awesome SDL* https://github.com/RobLoach/awesome-sdl

Milenociv, S. (2023). *Exploring the PC Game Engine Landscape* https://www.gamedeveloper.com/game-platforms/exploring-the-pc-game-engine-landscape

Unity Technologies. (2021). *GameObjects documentation* https://docs.unity3d.com/2020.1/Documentation/Manual/GameObjects.html

Unity Technologies. (n.d.a) *ECS concepts* https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html

Unity Technologies. (n.d.b). *Unity visual scripting* https://unity.com/features/unity-visual-scripting

Jackson, S. (n.d-a) *Getting started with MonoGame using XML* https://darkgenesis.zenithmoon.com/getting-started-with-monogame-using-xml.html

Jackson, S. (n.d-b). *MonoGame roundup for 2023* https://darkgenesis.zenithmoon.com/monogame-roundup-2023.html

Monogame. (n.d-a) *Monogame documentation* https://monogame.net/articles/tools/mgcb_editor.html

Monogame. (n.d-b) *Monogame showcase* https://monogame.net/showcase/

Nintendo. (n.d) *Super Mario Bros Gallery* https://www.nintendo.com/en-za/Games/NES/Super-Mario-Bros-803853.html - Gallery

PixiJs. (n.d) *PixiJS — The HTML5 Creation Engine* https://github.com/pixijs/pixijs

Rich, C. (n.d). *Basic Game Physics* https://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/D-Physics.pdf

SlashData. (2022). *State of the developer nation, 23rd edition*
https://docsend.com/view/4cmnvxa2xb5jd6hr?utm_source=SlashDataco&utm_medium=FreeReports_SoN23

SDL Wiki. (n.d) *Introduction to SDL 2.0*
https://wiki.libsdl.org/SDL2/Introduction

Terra, J. (2023). *Entity Component System: An Introductory Guide*
https://www.simplilearn.com/entity-component-system-introductory-guide-article

Villaneuva, A. (2024). *Elements of game engine*
https://www.outsourceaccelerator.com/articles/game-engine/

Orland, K. (2023). *Unity to devs: Pay up*
https://arstechnica.com/gaming/2023/09/game-developers-unite-against-unitys-new-per-install-pricing-structure/

Zarrand, A. (2018). *Game Engine Solutions*
Intech

Zenva. (2023). *What is Unity? – A Top Game Engine for Video Games*
https://gamedevacademy.org/what-is-unity/