TURUN YLIOPISTO
Kauppakorkeakoulu

Pelimoottorit ja sisällöntuotannon muutos: Siirtymä koodikeskeisyydestä generatiiviseen tekoälyyn

Tietojärjestelmätieteet
Kandidaatintutkielma

Laatija:
Juho Haaramäki

Ohjaaja:
FT Samuli Laato

19.12.2025
Turku

Opiskelijan lausunto tekoälyn käytöstä tähän tutkielmaan liittyen:

☐ **En ole käyttänyt tekoälyä hyödyntäviä työkaluja** tätä tutkielmaa kirjoittaessani.

☒ **Olen käyttänyt tekoälyä hyödyntäviä työkaluja** tätä tutkielmaa kirjoittaessani. Tämä käyttö on dokumentoitu tutkielman liitteessä. Vakuutan, että tekoälyä käytettiin yliopiston ohjeistuksen mukaisella tavalla.

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

Kandidaatintutkielma

Oppiaine: Tietojärjestelmätieteet
Tekijä: Juho Haaramäki
Otsikko: Pelimoottorit ja sisällöntuotannon muutos: Siirtymä koodikeskeisyydestä generatiiviseen tekoälyyn
Ohjaaja: FT Samuli Laato
Sivumäärä: 30 sivua
Päivämäärä: 19.12.2025

Tiivistelmä

Videopeliteollisuus on kasvanut merkittäväksi ohjelmistoteollisuuden alaksi, jonka tekniset vaatimukset ja kehitysmenetelmät eroavat perinteisestä ohjelmistotuotannosta. Pelikehitys on siirtynyt yhä enemmän koodikeskeisistä ohjelmointiympäristöistä kohti visuaalisia pelimoottoreita, jotka tarjoavat korkean abstraktiotason työkaluja monialaisten tiimien käyttöön. Tämän kandidaatintutkielman tavoitteena on tarkastella pelimoottoreita tietojärjestelmätieteen näkökulmasta ja selvittää, miten ne muuttavat ohjelmistokehityksen prosessia ja kehittäjän työnkuvaa erityisesti tekoälyintegraatioiden myötä.

Tutkielman keskeiset tutkimuskysymykset selvittävät, miten pelimoottoripohjainen kehitys eroaa perinteisestä ohjelmistokehityksestä työkalujen ja abstraktiotason näkökulmasta, sekä miten modernit tekoälytyökalut vaikuttavat sisällöntuotannon prosessiin. Tutkimus toteutettiin kirjallisuuskatsauksena, jossa analysoitiin ohjelmisto- ja peliohjelmistotekniikkaa käsittelevää tieteellistä kirjallisuutta. Lisäksi aineistona hyödynnettiin kolmen merkittävän pelimoottorin, Unityn, Unreal Enginen ja Godotin, teknistä dokumentaatiota tekoälyominaisuuksien vertailussa.

Tutkimuksen tulokset osoittavat, että pelimoottorit ovat hybridimallisia kehitysalustoja, jotka ratkaisevat pelikehitykselle tyypillisen koodin ja luovan työn välisen jaon tarjoamalla visuaalisia skriptaus- ja hallintatyökaluja perinteisen koodin rinnalle. Toisin kuin perinteiset ohjelmistokehykset, pelimoottorit piilottavat matalan tason teknisen toteutuksen valmiiden alijärjestelmien taakse, mikä mahdollistaa nopean iteratiivisen kehityksen mutta vähentää kehittäjän suoraa kontrollia järjestelmän toiminnasta.

Merkittävin tutkimuksessa havaittu murros liittyy tekoälyn rooliin: ala on siirtymässä deterministisestä, sääntöpohjaisesta sisällöntuotannosta kohti todennäköisyyksiin perustuvaa generatiivista tekoälyä. Tämä muuttaa kehittäjän roolia teknisestä toteuttajasta kohti sisällön kuraattoria ja kehotetekniikan osaajaa. Samalla se tuo ohjelmistotuotantoon uusia laadunvarmistuksen haasteita, sillä tekoälyn tuottamaa koodia ja sisältöä on vaikeampi varmistaa perinteisin menetelmin.

Avainsanat: pelikehitys, pelimoottori, tekoäly, generatiivinen tekoäly, proseduraalinen sisällönluonti, ohjelmistotuotanto, kehotetekniikka

SISÄLLYS

TAULUKOT

1 Johdanto

Videopeliteollisuus on kasvanut yhdeksi merkittävimmistä viihdeteollisuuden aloista ohittaen liikevaihdollaan jopa elokuva- ja musiikkiteollisuuden (Politowski ym., 2021). Nykyaikaiset videopelit voivat olla teknisesti monimutkaisia ohjelmistojärjestelmiä, joiden laajuus ja vaatimukset ovat kasvaneet merkittävästi viime vuosikymmeninä (Wang & Nordmark, 2015).

Perinteisesti ohjelmistokehitys on nojannut koodikeskeisiin menetelmiin (engl. *Code-Driven Development, CDD*) ja työkaluihin, kuten integroituihin kehitysympäristöihin (engl. *Integrated development environment, IDE*). Tässä tutkielmassa perinteisellä ohjelmistokehityksellä viitataan tähän koodikeskeiseen lähestymistapaan, jossa sovelluksen logiikka ja toiminnallisuus määritellään ensisijaisesti kirjoittamalla tekstimuotoista lähdekoodia integroiduissa kehitysympäristöissä (Chueca ym., 2023; Martínez ym., 2014). Pelikehityksen kasvanut monimutkaisuus ja tarve hallita visuaalisia elementtejä reaaliaikaisesti ovat kuitenkin johtaneet siirtymään kohti korkeamman abstraktiotason työkaluja, joita kutsutaan pelimoottoreiksi (engl. *Game engines*) (Agrahari & Chimalakonda, 2021).

Pelimoottorit, kuten Unity ja Unreal Engine, eivät ole vain koodieditoreja, vaan visuaalisia kehitysalustoja, jotka tarjoavat valmiit alijärjestelmät esimerkiksi fysiikanmallinnukseen, grafiikan renderöintiin ja äänen hallintaan (Ullmann ym., 2023). Nykyaikaiset pelimoottorit tarjoavat myös integroituja kehittäjäpalveluita, kuten pelianalytiikkaa, ja hyödyntävät tekoälyä esimerkiksi pelaajakokemuksen parantamiseen (Salama & Elsayed, 2021). Erityisesti proseduraalinen sisällöntuotanto (engl. *Procedural Content Generation, PCG*) on noussut keskeiseksi menetelmäksi, jolla pyritään ratkaisemaan sisällöntuotannon pullonkauloja automatisoimalla pelimaailman luontia algoritmien avulla (Risi & Preuss, 2020).

Vaikka videopelit ovat merkittävä osa ohjelmistoteollisuutta, pelikehityksen erityispiirteet ovat jääneet ohjelmistotuotannon tutkimuksessa vähemmälle huomiolle (Chen, 2025). Chueca ym. (2023) huomauttavat, että pelikehittäjät kohtaavat esimerkiksi koodin uudelleenkäytössä ja teknisen velan (engl. *technical debt*) hallinnassa haasteita, jotka eroavat perinteisestä ohjelmistokehityksestä. Erityisesti generatiivisen tekoälyn (engl. *generative artificial intelligence, GenAI*) integraatio pelimoottoreihin on luonut uudenlaisen toimintaympäristön, jonka vaikutuksia kehittäjän työnkulkuun ja prosessin hallintaan ei ole vielä kattavasti jäsennetty tietojärjestelmätieteen näkökulmasta (Chen, 2025). Hou ym. (2025) korostavat, että aikaisempi generatiivisen tekoälyn

tutkimus on keskittynyt liiaksi lopputuloksiin ja sivuuttanut ihmisen ja GenAI:n yhteistyön dynamiikan.

Tämän kandidaatintutkielman tavoitteena on tarkastella pelimoottoreita tietojärjestelmätieteen näkökulmasta kehitysalustoina. Tutkielma vertailee pelimoottoripohjaista kehitystä perinteiseen IDE-pohjaiseen ohjelmistokehitykseen ja selvittää, miten pelimoottorien tarjoamat tekoälytyökalut, erityisesti PCG, muuttavat kehitysprosessia.

Tutkielman tutkimuskysymykset ovat:

1. Miten pelimoottoripohjainen kehitys eroaa perinteisestä ohjelmistokehityksestä työkalujen ja abstraktiotason näkökulmasta?
2. Miten pelimoottoreiden tekoälytyökalut ja proseduraalisen sisällöntuotannon menetelmät vaikuttavat nykyaikaiseen pelikehityksen prosessiin?

Tutkielma toteutetaan kirjallisuuskatsauksena, ja sen aineisto koostuu ohjelmisto- ja peliohjelmistotekniikkaa, pelimoottoreita sekä tekoälyä käsittelevistä tieteellisistä artikkeleista sekä alan teknisestä kirjallisuudesta. Työn teoreettinen osuus perustuu vertaisarvioituun tieteelliseen kirjallisuuteen, kun taas kaupallisten pelimoottoreiden tekoälyratkaisujen kartoitus pohjautuu valmistajien tekniseen dokumentaatioon. On tiedostettava, että kaupallisen materiaalin luonne ja rajoitettu tieteellinen luotettavuus eroavat vertaisarvioidusta tiedosta. Valmistajien dokumentaatio kuitenkin tarjoaa ajantasaisimman ja yksityiskohtaisimman tiedon nopeasti kehittyvien kaupallisten työkalujen käytännön toteutuksista, mitä ei ole saatavilla perinteisessä tieteellisessä kirjallisuudessa. Ensimmäiseen tutkimuskysymykseen vastataan vertailemalla peliohjelmistotekniikan ja perinteisen ohjelmistokehityksen lähestymistapoja aiemman tutkimuskirjallisuuden pohjalta. Vertailussa keskitytään erityisesti siihen, miten siirtymä koodikeskeisestä työskentelystä mallipohjaiseen ja visuaaliseen kehittämiseen muuttaa kehittäjän roolia. Toiseen tutkimuskysymykseen vastataan kartoittamalla ja luokittelemalla nykyaikaisten pelimoottorien, kuten Unityn, Unreal Enginen ja Godotin, tarjoamia tekoälytyökaluja. Analyysissa eritellään sääntöpohjaiset menetelmät, kuten navigointi ja perinteinen PCG, ja uudet generatiivisen tekoälyn sovellukset sekä arvioidaan niiden vaikutusta sisällöntuotannon prosessiin. Lopuksi työn pohdintaosuudessa syntetisoidaan havainnot ja tarkastellaan tekoälyvetoisen kehityksen etuja ja haasteita.

2 Videopelien järjestelmäkehittäminen

2.1 Pelikehityksen erityispiirteet ja modernit haasteet

2.1.1 Tavoitteiden ja vaatimusten erityispiirteet

Pelikehitys eroaa perinteisestä ohjelmistotuotannosta pohjimmiltaan tuotteen käyttötarkoituksen ja tavoitteiden kautta. Kasurinen (2016) vertaa pelikehitystä perinteiseen ohjelmistotuotantoon ja toteaa, että niiden tavoitteet ovat usein päinvastaiset. Siinä missä perinteiset, prosessien tehostamiseen ja toiminnallisuuteen keskittyvät ohjelmistot pyrkivät minimoimaan ajan, joka käyttäjältä kuluu tietyn toiminnon suorittamiseen, pelien suunnittelussa tavoitteena on usein maksimoida tuotteen parissa vietetty aika.

Vaikka nykyään monet muutkin sovellukset, kuten sosiaalinen media, tavoittelevat käyttäjän sitouttamista, tämä ero korostuu erityisesti verrattaessa pelejä puhtaasti välineellisiin ohjelmistoihin, joiden arvo perustuu tehtävän nopeaan suorittamiseen. Koska pelien ensisijainen tarkoitus on viihdyttää, niiltä puuttuvat usein selkeät funktionaaliset vaatimukset (Wang & Nordmark, 2015). Sen sijaan kehitystä ohjaavat Wangin ja Nordmarkin (2015) esiin nostamat emotionaaliset vaatimukset tai Murphy-Hillin ym. (2014) kuvaama vaatimus hauskuudesta. Nämä tavoitteet heijastuvat suoraan kehittäjien kokemukseen, sillä sekä Pascarella ym. (2018) että Murphy-Hill ym. (2014) osoittavat kyselyissään, että pelikehittäjät kokevat projektien vaatimukset selvästi epäselvemmiksi kuin kollegansa muissa ohjelmistohankkeissa.

Peliala ei kuitenkaan ole täysin yhtenäinen tavoitteidensa suhteen. Vaikka valtaosa tuotannosta keskittyy viihteeseen, Murphy-Hill ym. (2014) nostavat esiin hyötypelit (engl. *serious games*), joita hyödynnetään esimerkiksi koulutuksessa, sotilassimulaatioissa ja lääketieteessä. Nämä sovellukset eroavat puhtaasta viihdekäytöstä, sillä ne mahdollistavat käyttäjille tuottavuuden tai oppimisen viihdearvon ohella (Murphy-Hill ym., 2014). Tällöin ne jakavat osittain Kasurisen (2016) määrittelemän hyötyohjelmiston tavoitteen ratkaista tietty ulkoinen tarve.

2.1.2 Tiimien rakenne ja resurssien erityispiirteet

Pelikehitys eroaa perinteisestä ohjelmistotuotannosta usein käsiteltävien resurssien jakauman suhteen. Eli siinä missä perinteiset ohjelmistoprojektit rakentuvat tyypillisesti lähdekoodin ympärille, monissa peliprojekteissa multimediasisällön rooli on korostunut(Pascarella ym., 2018).

Pascarella ym. (2018) vertailivat pelikehitystä perinteiseen ohjelmistokehitykseen analysoimalla 60 avoimen lähdekoodin projektin tiedostojakaumia sekä kyselytutkimuksella, johon vastasi 81 kehittäjää. Heidän aineistonsa osoitti selkeän eron tarkasteltujen projektien välillä. Perinteisissä ohjelmistoprojekteissa keskimäärin 80 prosenttia tiedostoista kuului niin kutsuttuun kehityskategoriaan (engl. *development category*), kun taas peliprojekteissa vastaava luku oli vain noin 30 prosenttia. Vaikka lähdekoodi on tämän kategorian suurin osatekijä, se sisältää myös muita teknisiä tiedostoja, kuten skriptejä, kirjastoja ja dokumentaatiota. (Pascarella ym., 2018)

Tämä resurssien jakauma heijastuu suoraan tiimien rakenteeseen ja asettaa haasteita eri ammattiryhmien väliselle yhteistyölle (Pascarella ym., 2018). Pelikehitys on luonteeltaan monialaista ja vaatii teknisen osaamisen lisäksi syvällistä osaamista esimerkiksi 3D-mallinnuksesta, äänisuunnittelusta ja käsikirjoittamisesta (Kasurinen, 2016; Wang & Nordmark, 2015). Pascarella ym. (2018) tunnistivat peliprojekteissa erikoistuneita alatiimejä, jotka keskittyvät yksinomaan multimediaresurssien hallintaan.

Tämä eriytyminen luo ilmiön, jota Wang ja Nordmark (2015) kuvaavat termillä "koodin ja luovan työn välinen jako" (engl. *code/art divide*). Kyse ei ole vain työnjaosta, vaan kulttuurisesta erosta. Murphy-Hillin ym. (2014) mukaan pelikehityksessä luovuutta arvostetaan usein teknistä kyvykkyyttä enemmän, mikä luo jännitteitä prosessiin. Tämä ilmenee siten, että teknisestä suunnittelusta ja arkkitehtuurista saatetaan tinkiä luovan vision toteuttamiseksi, mikä kasvattaa projektin teknistä velkaa ja johtaa kestävyydeltään heikkoihin ratkaisuihin. Koska tiimin jäsenet tulevat hyvin erilaisista taustoista, kyky kommunikoida ei-teknisten jäsenten kanssa on pelikehityksessä kriittisempi taito kuin perinteisessä ohjelmistotuotannossa. (Murphy-Hill ym., 2014)

Tiimien monimuotoisuus ulottuu myös organisaation johtotasolle. Murphy-Hill ym. (2014) huomauttavat, että pelialan johtotehtävissä toimii usein henkilöitä, joilla ei ole teknistä taustaa. Tämä voi johtaa tilanteisiin, joissa teknisten ongelmien, kuten bugien korjaamisen tai ylläpidon, merkitystä on vaikea perustella johdolle, joka tarkastelee tuotetta ensisijaisesti luovana kokemuksena (Murphy-Hill ym., 2014).

2.1.3 Prosessin epävakaus ja laadunvarmistuksen haasteet

Pelikehitystä leimaa korkea epävarmuus ja jatkuvat muutokset, sillä pelisuunnittelu ei ole koskaan lopullisesti valmis ennen toteutusta (Kasurinen, 2016). Suunnitelmia saatetaan muuttaa vielä

projektin myöhäisissäkin vaiheissa, mikä erottaa prosessin perinteisestä ohjelmistokehityksestä (Kasurinen, 2016). Murphy-Hill ym. (2014) selittävät tämän johtuvan siitä, että pelien keskeistä vaatimusta, hauskuutta, on vaikea todentaa ilman konkreettista kokeilua. Mikäli suunniteltu ominaisuus ei osoittaudu hauskaksi, se saatetaan hylätä kokonaan, mikä johtaa tehdyn työn hukkaan heittämiseen (Murphy-Hill ym., 2014). Wang ja Nordmark (2015) toteavat, että epärealistinen laajuus ja ominaisuuksien hallitsematon lisääntyminen ovat peliprojektien yleisimpiä ongelmia.

Jatkuva muutos vaikuttaa suoraan teknisiin ratkaisuihin ja arkkitehtuuriin. Pelikehittäjät välttävät usein raskasta etukäteissuunnittelua minimoidakseen turhan työn riskin (Murphy-Hill ym., 2014). Tämä johtaa Murphy-Hillin ym. (2014) mukaan usein "arkkitehtuurivelkaan", kun nopeita ratkaisuja tehdään pitkän tähtäimen ylläpidettävyyden kustannuksella. Vaikka peliala hyödyntää laajasti ketteriä menetelmiä, termiä ”Agile” käytetään Murphy-Hill ym. (2014) haastateltavien mukaan toisinaan vain kuvaamaan prosessin puutetta.

Merkittävin tekninen seuraus prosessin epävakaudesta on testauksen haasteellisuus. Pascarella ym. (2018) osoittivat kyselytutkimuksessaan, että pelikehittäjät hyödyntävät yksikkötestausta tilastollisesti merkittävästi vähemmän kuin muut ohjelmistokehittäjät. Tutkijat päättelivät tämän johtuvan muun muassa testien kirjoittamisen haasteellisuudesta pelikontekstissa. Automaation vähäisen käytön on argumentoitu johtuvan useasta tekijästä. Pelien tila-avaruus on liian laaja kattavaan testaukseen, ja pelilogiikkaa on vaikea erottaa käyttöliittymästä. Lisäksi automatisoidut testit ovat alttiita rikkoutumaan, kun pelisuunnittelija muuttaa pelimekaniikkaa. Näistä syistä peliteollisuus nojaa edelleen vahvasti manuaaliseen testaukseen, jota pidetään automaatiota joustavampana ratkaisuna muuttuvassa ympäristössä. (Murphy-Hill ym., 2014)

2.2 Pelikehityksen teknologiset ympäristöt

Edellä kuvatut pelikehityksen erityispiirteet, kuten monialaisten tiimien tarpeet ja prosessin iteratiivisuus, asettavat kehitystyökaluille vaatimuksia, joihin perinteiset ratkaisut eivät välttämättä yksinään pysty vastaamaan. Videopelien kehittäminen vaatii ohjelmistotyökaluja, jotka kykenevät hallitsemaan paitsi sovelluslogiikkaa, myös pelien vaatimaa laajaa multimediasisältöä (Pascarella ym., 2018).

Kehitystyössä voidaan tunnistaa kaksi pääasiallista lähestymistapaa, perinteinen koodikeskeinen (engl. code-centric) kehitys integroiduissa kehitysympäristöissä (engl. Integrated Development Environment, IDE) ja moderni visuaalinen kehitys pelimoottoreissa (Martínez ym., 2014). Vaikka

nämä ympäristöt jakavat yhteisiä ohjelmistoteknisiä piirteitä, kuten koodieditoinnin ja virheenkorjauksen, ne eroavat toisistaan merkittävästi abstraktiotason ja resurssien hallinnan suhteen (Cai, 2015; Wang & Nordmark, 2015).

Agrahari ja Chimalakonda (2021) määrittelevät pelimoottorit laajoiksi kehityspaketeiksi, jotka vähentävät matalan tason ohjelmointityötä. IDE-ympäristöt taas tarjoavat suoremman kontrollin lähdekoodiin (Martínez ym., 2014).

2.2.1 Perinteiset ohjelmistokehitysympäristöt

Perinteisessä ohjelmistokehityksessä työn keskiössä ovat integroidut kehitysympäristöt, jotka yhdistävät useita eri työkaluja yhdeksi graafiseksi käyttöliittymäksi. Näiden ympäristöjen päätavoitteena on tehostaa ohjelmoijan työtä ja vähentää kognitiivista kuormaa keskittämällä työkalut yhteen näkymään (Gasparic ym., 2017).Vaikka tavoitteena on työn sujuvoittaminen, ohjelmistojen monimutkaisuus on kasvattanut kehittäjien informaatiotarpeita, jolloin perinteinen IDE ei välttämättä enää riitä. Cai (2015) huomauttaa, että kehittäjät joutuvat usein vaihtamaan kontekstia koodinäkymän ja muiden visuaalisten apuvälineiden tai sovellusten välillä, mikä voi katkaista kehittäjän ajatuskulun ja heikentää tehokkuutta. Tämä kognitiivinen haaste korostuu pelikehityksessä, jossa visuaalisen ja toiminnallisen palautteen tarve on jatkuvaa.

Tämä lähestymistapa edustaa Martínezin ym. (2014) määrittelemää koodikeskeistä paradigmaa, jossa sovelluksen logiikka ja rakenne määritellään ensisijaisesti tekstimuotoisessa lähdekoodissa. Tässä ympäristössä kehittäjän vuorovaikutus järjestelmän kanssa tapahtuu abstraktien tekstikomentojen kautta, mikä eroaa merkittävästi mallipohjaisesta kehityksestä, jossa abstraktiotaso on korkeammalla ja artefakteja (engl. *artifact*) hallitaan visuaalisesti (Ricca ym., 2018). Ricca ym. (2018) havaitsivat vertailututkimuksessaan, että puhtaasti koodikeskeinen lähestymistapa saattaa olla hitaampi ylläpitotehtävissä verrattuna mallipohjaisiin menetelmiin, sillä koodin ymmärtäminen ja navigointi vaativat enemmän aikaa kuin visuaalisten mallien käsittely. Tämä havainto tukee Cain (2015) argumenttia siitä, että pelkkä tekstipohjainen käyttöliittymä ei vastaa tehokkaasti monimutkaisiin tiedonhakutarpeisiin.

Pascarellan ym. (2018) havainto peliprojektien tiedostojakauman painottumisesta ei-tekstuaalisiin elementteihin tukee näkemystä siitä, että pelkkä koodikeskeinen näkymä ei riitä pelikehityksen tarpeisiin. Tämä sokea piste korostuu, kun otetaan huomioon Martínezin ym. (2014) havainto koodikeskeisyydestä.

Visuaalisen palautteen puute ja raskaat prosessit ovat ristiriidassa pelikehityksen iteratiivisen luonteen kanssa. Murphy-Hill ym. (2014) korostavat, että pelien tärkein vaatimus on hauskuus, joka on subjektiivinen ominaisuus ja löytyy usein vain nopean kokeilun kautta. Perinteinen IDE-työskentely nojaa kuitenkin raskaaseen "muokkaa-rakenna-suorita" -sykliin, joka hidastaa tätä luovaa prosessia (Murphy-Hill ym., 2014). Ricca ym. (2018) tukevat tätä näkemystä osoittamalla, että koodikeskeinen työskentelytapa on vähemmän tehokas tilanteissa, joissa vaaditaan nopeaa muutosten tekemistä ja vaikutusten arviointia.

Näistä rajoitteista huolimatta perinteiset kehitysympäristöt ovat säilyttäneet asemansa pelikehityksessä, mutta niiden rooli on erikoistunut. IDE on edelleen toimiva työkalu silloin, kun vaaditaan tarkkaa kontrollia matalan tason toiminnoista, kuten muistinhallinnasta tai optimoinnista (Wang & Nordmark, 2015). Tämän vuoksi peliteollisuus on omaksunut hybridimallin, jossa työnkuvat ja työkalut eriytyvät. Kuten aiemmin on mainittu, Pascarellan ym. (2018) mukaan peliprojekteissa voidaan tunnistaa erikoistuneita alatiimejä, joista toiset keskittyvät sisällöntuotantoon visuaalisilla työkaluilla, kun taas ohjelmoijat hyödyntävät perinteisiä IDE-ympäristöjä arkkitehtuurin ja suorituskyvyn hallintaan. Tämä työnjako mahdollistaa sekä koodikeskeisen tarkkuuden (Martínez ym., 2014), että visuaalisen kehityksen nopeuden (Ricca ym., 2018) hyödyntämisen samassa projektissa.

2.2.2 Pelimoottorin arkkitehtuuri ja luovan työn tuki

Pelimoottorit voidaan määritellä laajoiksi ohjelmistokehityspaketeiksi, joiden keskeinen tavoite on vähentää matalan tason ohjelmointityötä tarjoamalla valmiita ratkaisuja yleisiin kehitystehtäviin (Agrahari & Chimalakonda, 2021). Politowski ym. (2021) tarkastelivat pelimoottorien luonnetta vertailemalla 282 avoimen lähdekoodin pelimoottoria ja 282 ohjelmistokehystä sekä haastattelemalla 124 pelimoottorikehittäjää. Heidän tutkimuksensa osoitti, että vaikka pelimoottorit jakavat teknisesti monia piirteitä perinteisten ohjelmistokehysten (engl. *Software frameworks)* kanssa, kehittäjät mieltävät ne usein erillisiksi, kokonaisvaltaisemmiksi alustoiksi, jotka mahdollistavat paremman hallinnan kehitysympäristöstä. Käytännössä tämä tarkoittaa, että moottorit tarjoavat integroidut alijärjestelmät esimerkiksi fysiikanmallinnukseen, tekoälyn toteutukseen ja pelimaailman hallintaan, jolloin kehittäjien ei tarvitse rakentaa näitä perusmekaniikkoja tyhjästä (Agrahari & Chimalakonda, 2021).

Pelimoottorien tarjoama modulaarinen arkkitehtuuri tukee kehittäjiä monimutkaisten ohjelmistoprojektien hallinnassa (Ullmann ym., 2023). Ullmann ym. (2023) jakavat pelimoottorit itsenäisiin alijärjestelmiin, joista jokainen vastaa tietystä toiminnallisuudesta, kuten renderöinnistä tai äänestä. Tämä modulaarisuus ei ainoastaan selkeytä ohjelmiston rakennetta, vaan mahdollistaa myös eri vastuualueiden eriyttämisen kehitystiimin sisällä. Politowski ym. (2021) tukevat tätä näkemystä rinnastamalla pelimoottorit ohjelmistokehyksiin. Moottorit tarjoavat standardoidun ja laajennettavan perustan, joka abstrahoi tekniset yksityiskohdat ja vähentää päällekkäisen työn määrää peliprojektien välillä (Politowski ym., 2021).

Pascarella ym. (2018) ovat osoittaneet, että peliprojekteissa valtaosa tiedostoista on multimediaa, kuten grafiikkaa ja ääntä, eikä lähdekoodia. Käytännössä pelimoottori toimii keskitettynä alustana, joka yhdistää projektin resurssit toimivaksi ohjelmistoksi (Politowski ym., 2021). Politowski ym. (2021) huomauttavatkin, että pelimoottorien kehitys on nimenomaan perustunut tarpeeseen hallita pelikoodia ja kehitysympäristöä, mikä mahdollistaa nykyaikaisten, monitieteisten tiimien tehokkaan työskentelyn.

Pelimoottorien arkkitehtuuri toimii perustana *Code/Art Divide* -ongelman ratkaisemiselle eli teknisen ja luovan työn yhdistämiselle. Moottorit tarjoavat korkean tason abstraktion, joka piilottaa teknisen monimutkaisuuden graafisten käyttöliittymien taakse. (Wang & Nordmark, 2015) Käytännön tasolla tämä näkyy visuaalisena skriptauksena. Salama ja Elsayed (2021) toteavat, että tällaiset järjestelmät mahdollistavat pelilogiikan nopean kehittämisen ilman, että kehittäjän tarvitsee kirjoittaa C++-koodia. Tämä on merkittävää, sillä C++-kielen käyttö vaatii enemmän ohjelmointikokemusta kuin esimerkiksi C#- tai JavaScript-kielten käyttö (Salama & Elsayed, 2021).

Pelimoottorien kehitys on muuttanut peliteollisuuden taloudellisia rakenteita merkittävästi. Pelimoottoreiden lisensointimallit ovat pudottaneet kehitystyön aloituskustannukset murto-osaan aiemmasta.(Torres-Ferreyros ym., 2016) Tämä on avannut markkinoita myös itsenäisille kehittäjille (engl. *Indie developers*), sillä pelimoottorit ovat madaltaneet aloituskynnystä merkittävästi abstrahoimalla teknisen toteutuksen (Politowski ym., 2021). Wang ja Nordmark (2015) tuovat esille, että nykyaikaiset moottorit tukevat emergenttiä toimintaa, jossa objektit vuorovaikuttavat keskenään sääntöjen pohjalta. Tämä dynaamisuus luo teknisen perustan Risi ja Preussin (2020) määrittelemälle modernille pelitekoälylle, joka kattaa myös automaattisen sisällöntuotannon kaltaisia laajoja sovellusalueita.

3 Pelimoottorien tekoälysovellukset

Tässä luvussa pelimoottorien tekoälyratkaisuja tarkastellaan kolmen pelimoottorin kautta: Unity, Unreal Engine ja Godot. Nämä on valittu edustamaan pelikehityksen nykytilaa siten, että Unity ja Unreal Engine toimivat esimerkkeinä laajasti käytetyistä kaupallisista ekosysteemeistä, kun taas Godot edustaa avoimen lähdekoodin lähestymistapaa.

3.1 Tekoäly osana pelimoottorin arkkitehtuuria

Nykyaikaiset pelimoottorit ovat kehittyneet graafisista renderöintialustoista laajoiksi ohjelmistokehitysympäristöiksi, jotka sisältävät useita erikoistuneita alijärjestelmiä (Ullmann ym., 2023). Tämä modulaarinen rakenne mahdollistaa sen, että kehittäjät voivat hyödyntää valmiita ratkaisuja monimutkaisten tekoälytoimintojen, kuten reitinhaun tai päätöksenteon, toteuttamiseen ilman, että niitä tarvitsee rakentaa alusta asti (Agrahari & Chimalakonda, 2021).

Historiallisesti pelitekoäly on ymmärretty eri tavalla kuin akateeminen tekoälytutkimus. Yannakakis ja Togelius (2018) huomauttavat, että peliteollisuus on perinteisesti keskittynyt NPC eli ei-pelaajahahmojen (engl. *Non-player character, NPC*) uskottavan käytöksen luomiseen deterministisillä menetelmillä, kuten äärellisillä tilakoneilla (engl. *Finite State Machines*, *FSM*) ja hakualgoritmeilla. Tavoitteena on ollut luoda immersiivinen kokemus pikemminkin kuin optimaalinen älykkyys (Risi & Preuss, 2020).

Viime vuosina ala on kuitenkin murroksessa. Risi ja Preuss (2020) toteavat, että pelitekoälyn määritelmä on laajentunut kattamaan pelkän pelaamisen lisäksi myös sisällöntuotannon ja pelianalytiikan. Nykyaikaiset pelimoottorit eivät enää tarjoa vain staattisia työkaluja, vaan integroivat yhä syvemmin koneoppimista ja dynaamisia järjestelmiä osaksi kehitysprosessia, mikä mahdollistaa täysin uudenlaisten pelityyppien luomisen (Risi & Preuss, 2020).Tämä siirtymä näkyy siinä, kuinka moottorit tarjoavat yhä korkeamman tason abstraktioita tekoälyn toteuttamiseen, mahdollistaen monimutkaiset simulaatiot ja mukautuvat pelikokemukset (Politowski ym., 2021; Risi & Preuss, 2020).

3.2 NPC-logiikka ja päätöksentekojärjestelmät

NPC-hahmojen uskottava toiminta edellyttää järjestelmää, joka ohjaa niiden päätöksentekoa loogisesti. Kehittäjät ovat perinteisesti mallintaneet hahmojen tiloja, kuten "etsi" tai "hyökkää", sekä siirtymiä näiden välillä käyttämällä graafisia tilamalleja tai modulaarisia puurakenteita (engl. *Behavior Trees*). Jälkimmäinen kehitettiin ratkaisemaan tilakoneiden ylläpito-ongelmia, ja se

mahdollistaa monimutkaisten käyttäytymismallien rakentamisen yhdistelemällä pienempiä tehtävä- ja kontrollisolmuja. (Yannakakis & Togelius, 2018)

Nykyaikaiset pelimoottorit eroavat merkittävästi siinä, miten ne tarjoavat työkaluja näiden logiikoiden toteuttamiseen. Unreal Engine 5:ssä päätöksentekoa mallinnetaan visuaalisella *Behavior Tree* -työkalulla, joka on integroitu syvälle moottorin arkkitehtuuriin. Järjestelmä erottaa päätöksentekologiikan ja sensoritiedon toisistaan *Blackboard*-arkkitehtuurin avulla. *Blackboard*-arkkitehtuuri toimii jaettuna muistina, josta tekoäly voi lukea havaintojaan. (Epic Games, 2025a) Lisäksi Unreal tarjoaa suorituskykyoptimoidun hybridimallin nimeltä *State Tree*, joka on kehitetty erityisesti suurten hahmojoukkojen hallintaan (Epic Games, 2025d).

Unity on historiallisesti tukeutunut ulkoisiin lisäosiin, mutta on tuonut Unity 6 -versiossa kehittäjille oman visuaalisen *Unity Behavior* -työkalunsa. Työkalun suunnittelussa on painotettu ihmisluettavuutta, mutta sen teknisenä tavoitteena on tehostaa pelin suorituskykyä hyödyntämällä tapahtumapohjaista (engl. *Event-driven*) arkkitehtuuria, jolloin tekoälyn ei tarvitse tarkistaa tilaansa jatkuvasti, vaan se reagoi ainoastaan muutoksiin. (Unity Technologies, 2025d)

Godot 4 edustaa puolestaan joustavampaa filosofiaa, jossa kehittäjälle ei tarjota yhtä sitovaa ratkaisua. Moottori tarjoaa modulaarisen alustan, joka ei sido kehittäjää yhteen visuaaliseen editoriin, vaan mahdollistaa logiikan rakentamisen joko kirjoittamalla koodia tai integroimalla yhteisön luomia lisäosia. Suorituskyvyn osalta Godot hyödyntää palvelinpohjaista *NavigationServer*-arkkitehtuuria, joka erottaa raskaan laskennan pelilogiikasta. Godot tarjoaa myös joustavan *AnimationTree*-tilakonejärjestelmän, jonka avulla voidaan animaatioiden luomisen lisäksi ohjata pelilogiikkaa. (Godot Engine, 2025a; Godot Engine, 2020)

3.3 Navigointi ja joukkoäly

NPC-hahmojen älykäs liikkuminen pelimaailmassa perustuu karttamaisiin tietorakenteisiin (engl. *Navigation Mesh*, NavMesh), jotka määrittelevät kuljettavissa olevat alueet (Yannakakis & Togelius, 2018). Tässä menetelmässä pelimaailman lattiapinta jaetaan geometrisiin monikulmioihin, mikä tekee siitä laskennallisesti tehokkaan verrattuna perinteiseen ruudukkoon. Navigaatioverkko muodostaa graafin, jonka solmujen välillä lyhin tai kustannustehokkain reitti lasketaan hakualgoritmilla, tyypillisesti A*-algoritmilla (Yannakakis & Togelius, 2018).

Unreal Engine 5 ratkaisee suurten väkijoukkojen simuloinnin hyödyntämällä dataorientoitunutta (engl. *data-oriented*) *MassEntity*-arkkitehtuuria (Epic Games, 2025f). Perinteisen raskaan reitinhakulaskennan sijaan suuriin väkijoukkoihin sovelletaan kevyempää, virtaukseen perustuvaa *Zone Graph* -menetelmää (Epic Games, 2025g). Lisäksi moottori käyttää erillistä *Mass Avoidance* -laskentajärjestelmää, joka on suunniteltu estämään hahmojen törmäilyn toisiinsa kuormittamatta pelin fysiikkamoottoria, mahdollistaen kymmenien tuhansien hahmojen samanaikaisen toiminnan (Epic Games, 2025h).

Godot 4 on uudistanut navigaatiojärjestelmänsä palvelinpohjaiseksi, mikä parantaa pelin vakautta. Tämä ratkaisu siirtää reitinhakulaskennan taustalle *NavigationServer*-arkkitehtuuriin, jolloin raskaat laskutoimitukset eivät pysäytä pelin päätoimintoja (Godot Engine, 2025d; Godot Engine, 2020). Godot hyödyntää myös edistynyttä RVO-väistelyalgoritmia (engl. *Reciprocal Velocity Obstacles*), joka mahdollistaa agenttien dynaamisen liikkumisen ilman törmäyksiä (Godot Engine, 2025e). Lisäksi järjestelmä tukee navigaatiokartan päivittämistä reaaliajassa pelimaailman muuttuessa, mikä on kriittistä dynaamisissa peliympäristöissä (Godot Engine, 2025f).

Unityn lähestymistapa korostaa helppokäyttöisyyttä ja modulaarisuutta *AI Navigation* -ohjelmistopaketin kautta. Toisin kuin perinteiset staattiset ratkaisut, Unityn tarjoamat komponentit, kuten *NavMesh Surface*, mahdollistavat navigaatioverkon luomisen ja päivittämisen dynaamisesti pelin ajon aikana. Tämä on erityisen merkittävää proseduraalisesti luoduissa kentissä, joissa liikkumisalueet eivät ole tiedossa etukäteen. Lisäksi *NavMesh Link* -komponentti ratkaisee monitasoisten kenttien haasteet mahdollistamalla agenttien hyppäämisen erillisten navigaatioverkkojen välillä ilman monimutkaista koodaamista. (Unity Technologies, 2025a; 2025b)

3.4 Generatiivinen tekoäly ja automaattinen sisällönluonti

Pelikehityksen sisällöntuotanto on historiallisesti kehittynyt manuaalisesta käsityöstä kohti yhä korkeampaa automaatiota. Perinteisesti tätä tarvetta on ratkaistu proseduraalisella sisällöntuotannolla (*Procedural Content Generation, PCG*), jossa pelimaailmoja ja objekteja luodaan algoritmisesti ennalta määriteltyjen sääntöjen ja parametrien pohjalta. Nämä menetelmät ovat luonteeltaan deterministisiä, eli samat syöteparametrit tuottavat aina saman lopputuloksen, mikä takaa kehittäjälle tarkan kontrollin. (Yannakakis & Togelius, 2018)

Generatiivisen tekoälyn (engl. *Generative Artificial Intelligence, GenAI*) yleistyminen on kuitenkin tuonut rinnalle uuden paradigman, jota Hassan ym. (2024) kutsuvat termillä FMware (*Foundation*

Model-based software). Tällä tarkoitetaan laajoilla datamassoilla esikoulutettujen perusmallien (*engl. foundation models*), kuten suurten kielimallien (*Large Language Models, LLM*), päälle rakentuvia ohjelmistoja, joiden toiminta perustuu deterministisiten sääntöjen sijaan stokastisiin todennäköisyyksiin (Hassan ym., 2024). Chen (2025) soveltaa tätä käsitettä pelikehitykseen luokittelemalla GenAI-vetoiset pelit FMwareksi, mikä merkitsee siirtymää koodatusta logiikasta epädeterministiseen luomiseen ja tuo mukanaan uusia haasteita esimerkiksi laadunvarmistukselle. Tämä teknologinen murros muuttaa myös pelikehittäjän työnkuvaa, siirtäen painopistettä koodaamisesta kohti kehotetekniikkaa (*engl. Prompt Engineering*), jossa kehittäjä ohjaa tekoälyä luonnollisella kielellä (Rozo-Torres & Sarmiento, 2024).

3.4.1 Sääntöpohjaisuudesta todennäköisyyksiin

Sisällöntuotannon tekninen perusta on jakautumassa kahteen selvästi erottuvaan lähestymistapaan: Perinteiseen sääntöpohjaiseen konstruktioon ja tekoälymalleihin perustuvaan ennustamiseen.

Perinteiset, kaupallisissa peleissä yleisesti käytetyt PCG-menetelmät ovat usein luonteeltaan konstruktiivisia (engl. *constructive*). Tämä tarkoittaa, että sisältö luodaan yhdessä vaiheessa sääntöjen ja algoritmien perusteella ilman erillistä, generoinnin aikaista arviointi- ja korjausprosessia (Lazaridis ym., 2022). Esimerkkinä tästä toimii Lazaridisin ym. (2022) kehittämä *Spawn Algorithm*, joka on suunniteltu generoimaan 2D-pelikarttoja. Algoritmi ei arvaa, vaan suorittaa toiminnon ennalta määritellyn logiikan mukaan. Esimerkiksi suuret huoneet sijoitetaan ensin tiettyjen kriteerien mukaan, minkä jälkeen tyhjät tilat täytetään. (Lazaridis ym., 2022)

Käytännön tasolla tämä konstruktiivinen menetelmä on integroitu suoraan nykyaikaisiin kehitystyökaluihin. Esimerkiksi Unreal Engine 5:n PCG Framework edustaa graafipohjaista (*engl. node-based*) arkkitehtuuria, jossa sisältöä ei luoda tyhjästä, vaan prosessoimalla dataa (Epic Games, 2025e). Tämä työkalu perustuu pisteiden näytteistämiseen (*engl. sampling*) pelimaailman geometriasta ja niiden suodattamiseen tarkkojen sääntöjen perusteella. Kehittäjä voi esimerkiksi rakentaa logiikan, joka poistaa automaattisesti puuston jyrkiltä rinteiltä tai rakennusten sisältä, mutta säilyttää tarkan kontrollin lopputuloksesta. Koska prosessi on deterministinen, sama syöte (engl. *seed*) tuottaa aina identtisen lopputuloksen. (Epic Games, 2025e)

Generatiivinen tekoäly tuo konstruktiivisen menetelmän rinnalle uudenlaisen toimintaperiaatteen, jota Hassan ym. (2024) kuvaavat termillä ”FMware”. Tässä mallissa ohjelmiston logiikkaa ei koodata eksplisiittisinä sääntöinä, vaan toiminnallisuus rakentuu esikoulutettujen perusmallien tuottamiin todennäköisyyksiin.

Pelimoottoritasolla tämä teoreettinen malli konkretisoituu esimerkiksi Unityn Sentis -kehyksessä, joka on tarkoitettu neuroverkkojen ajamiseen reaaliaikaisesti suoraan loppukäyttäjän laitteella, mitä kutsutaan usein ajonaikaiseksi tekoälyksi (engl. *Runtime AI*). Ratkaisu mahdollistaa esimerkiksi älykkäiden NPC-hahmojen päätöksenteon tai dynaamisen sisällön luomisen pelitapahtuman aikana ilman pilviyhteyttä. (Unity Technologies, 2025e) Unity Sentis toimii Hassanin kuvaaman FMware-mallin teknisenä toteuttajana, joka mahdollistaa esikoulutettujen mallien, kuten suurten kielimallien, suorittamisen osana reaaliaikaista pelilogiikkaa.

Järjestelmän ytimessä on ONNX-standardi, joka toimii eräänlaisena yleiskielenä eri tekoälymalleille. Sen sijaan, että pelimekaniikka rakennettaisiin perinteisillä "jos–niin"-säännöillä, peliin tuodaan esikoulutettu keinotekoinen neuroverkko (engl. *Artificial neural network, ANN*). Yannakakis ja Togelius (2018) kuvaavat neuroverkkoa biologiasta inspiroituneeksi järjestelmäksi, jossa tietoa ei tallenneta sääntöihin, vaan ”neuronien” välisten yhteyksien painokertoimiin. Matemaattisesti kyseessä on funktio, joka laskee syötteille, esimerkiksi pelitilanteelle, painotetun summan ja ajaa sen aktivaatiofunktion läpi tuottaakseen ennusteen (Yannakakis & Togelius, 2018).

Pelimoottori syöttää tälle mallille pelitilanteen dataa numeerisessa muodossa, ja malli ennustaa todennäköisimmän lopputuloksen ilman, että kehittäjä on erikseen ohjelmoinut kyseistä reaktiota koodiin (Unity Technologies, 2025e). Koska prosessi perustuu todennäköisyyksien optimointiin eikä ehdottomiin sääntöihin, lopputuloksen tarkka syntymekanismi jää kehittäjältä piiloon tavalla, jota perinteisessä ohjelmoinnissa ei tapahdu.

Vaikka Unreal Engine painottaa sisällöntuotannossa sääntöpohjaista PCG-työkalua, moottori tukee myös neuroverkkojen päättelyä (engl. *Neural Network Inference*, *NNI*) (Epic Games ,2025c). Toisin kuin Unityn Sentis, jota markkinoidaan yleiskäyttöisenä ratkaisuna pelilogiikkaan (Unity Technologies, 2025e), Unrealin NNI on integroitu tiukemmin grafiikkaputkeen. Sen merkittävin sovellus on ML Deformer, joka käyttää neuroverkkoa reaaliaikaiseen hahmomallien simulaatioon (Epic Games, 2025b). Tämä heijastaa moottorien strategista eroa. Unity tuo neuroverkot osaksi pelimekaniikkaa (Unity Technologies, 2025e), kun taas Unreal valjastaa ne ensisijaisesti visuaalisuuden ja simulaation tarkkuuden parantamiseen (Epic Games, 2025b; 2025c).

Hassan ym. (2024) korostavat, että siinä missä perinteinen algoritmi on suunniteltu välttämään virheet noudattamalla rajoitteita, ennustava FMware-malli optimoi todennäköisyyttä eikä totuutta. Tämän seurauksena järjestelmä voi tuottaa uskottavan näköistä mutta faktuaalisesti virheellistä sisältöä, mitä kutsutaan hallusinoinniksi (engl. *hallucination*).

Pelimoottorien kehitysympäristössä tämä näkyy kontrollin heikkenemisenä. Jos graafipohjainen sääntö tuottaa virheen, kehittäjä voi jäljittää ja korjata kyseisen solmun. Sen sijaan inferenssipohjaisessa järjestelmässä, kuten Unity Sentisissä, vastaava virhe syntyy neuroverkon sisäisessä prosessissa, jota ei voi suoraan muokata. Tämä on tunnistettu merkittäväksi haasteeksi ohjelmistoalalla ja Hou ym. (2025) toteavatkin, että mallien epäluotettavuus ja taipumus hallusinointiin tekevät perinteisestä testauksesta ja varmistamisesta vaikeaa, sillä tuotettu koodi tai sisältö voi olla uskottavaa mutta virheellistä. Pelikehityksessä Chen (2025) tiivistää tämän laadunvarmistuksen ongelmaksi. Kun lopputulos ei ole enää sääntöjen takaamaa vaan vaihtelee tilastollisesti, tasalaatuisuuden ylläpitäminen vaatii uudenlaisia työtapoja.

3.4.2 Työnkulun muutos ja uudet osaamisvaatimukset

Teknologinen murros heijastuu suoraan kehittäjän päivittäiseen työhön ja työkaluihin. Hassan ym. (2024) kuvaavat tätä siirtymänä, jossa ihmisen rooli muuttuu teknisestä toteuttajasta tavoitteen määrittelijäksi. Kehittäjä ei enää kirjoita jokaista loogista askelta itse, vaan kuvailee halutun lopputuloksen.

Pelimoottoreissa tämä muutos konkretisoituu Unityn Muse -tekoälyalustassa, joka jakaa toiminnallisuudet sisällöntuotannon generaattoreihin ja älykkääseen avustajaan. Generaattorien avulla monimutkainen tekninen toteutus korvautuu kuvailulla. Esimerkiksi tekstuurigeneraattori (engl. *Texture generator*) luo tekstikomennon perusteella PBR-materiaalit (engl. *Physically Based Rendering*) värikarttoineen, normaalikarttoineen ja karheusdatoineen, ilman että kehittäjän tarvitsee säätää parametreja käsin. Prosessin ytimessä on Unity Assistant -työkalu, joka eroaa yleisistä kielimalleista ymmärtämällä projektin kontekstin. Valmistajan mukaan se kykenee suorittamaan hakuja suoraan editorin sisällä ja kirjoittamaan C#-koodia, joka on välittömästi yhteensopivaa kyseisen peliprojektin kanssa. (Unity Technologies, 2025c) Vaikka ominaisuus on lupaava, sen todellinen luotettavuus laajoissa ja monimutkaisissa tuotantoprojekteissa jää vielä nähtäväksi.

Tämän vuoksi kehittäjän ydinosaaminen on muuttumassa. Rozo-Torres ja Sarmiento (2024) toteavat, että ala on siirtymässä perinteisistä koodaustiedoista kohti taitoja, joissa korostuu kyky kommunikoida tekoälyn kanssa luonnollisella kielellä. Hou ym. (2025) kuitenkin varoittavat tekoälyn epädeterministisestä luonteesta. Koska sama kehote voi tuottaa eri suorituskerroilla täysin erilaisen lopputuloksen, järjestelmän luotettavuuden varmistaminen vaikeutuu (Hou ym., 2025). Pelimoottorin tekoälyavustajan, kuten Unity Assistantin, etu on kuitenkin juuri niiden kyky hyödyntää projektikohtaista kontekstitietoa, mikä asettaa mallille tiukemmat rajoitteet ja auttaa vähentämään lopputulosten satunnaisuutta (Unity Technologies, 2025c). Kehotetekniikka ei siis ole

vain passiivista pyytämistä, vaan se vaatii kehittäjältä kykyä pilkkoa ongelmat osiin ja validoida vaihtelevat vastaukset kriittisesti. Kehittäjästä tulee tällöin kuraattori, jonka tärkein tehtävä on arvioida ja integroida tekoälyn tuottamat palaset toimivaksi ja yhtenäiseksi kokonaisuudeksi. (Hou ym., 2025; Rozo-Torres & Sarmiento, 2024)

3.5 Yhteenveto ja vertailutaulukko

Edellä käsitellyt tekoälysovellukset osoittavat, että pelimoottorit eivät ole enää pelkkiä suoritusympäristöjä, vaan ne kykenevät yhä itsenäisempään pelilogiikan suorittamiseen ja sisällön luomiseen. Vaikka Unity, Unreal Engine ja Godot jakavat samat perusperiaatteet esimerkiksi navigaation ja päätöksenteon osalta, niiden arkkitehtuuriset valinnat ja kehitysfilosofiat kuitenkin eroavat toisistaan.

Alla oleva pelimoottoreiden ominaisuuksien vertailu (Taulukko 1) perustuu Unityn, Epic Gamesin ja Godotin tarjoamaan tekniseen dokumentaatioon.

Taulukko 1. Pelimoottoreiden tekoälyominaisuudet

Ominaisuus/ Pelimoottori	Unity	Unreal Engine	Godot
Päätöksenteko ja logiikka	**Unity Behavior:** Visuaalinen, tapahtumapohjainen järjestelmä. Korostaa helppokäyttöisyyttä. Perustuu deterministiseen sääntölogiikkaan ja käyttäytymispuihin	**Behavior Tree ja State Tree:** Vakiintunut Behavior Tree -järjestelmä ja sen rinnalla kevyt State Tree -hybridimalli suurten massojen hallintaan. Perustuu deterministisiin sääntöihin ja hierarkkisiin tilakoneisiin	**Modulaarinen:** Ei sitovaa mallia. Logiikka koodilla tai lisäosilla. AnimationTree toimii tilakoneena. Perustuu deterministiseen sääntölogiikkaan
Navigointi ja väkijoukot	**AI Navigation:** NavMesh Surface tukee dynaamista päivitystä. NavMesh Link yhdistää erilliset alueet. Perustuu A*-hakualgoritmiin.	**MassEntity & Zone Graph:** Data-orientoitunut simulaatio ja virtauspohjainen navigaatio suurille joukoille. Perustuu A*-algoritmiin.	**NavigationServer:** Palvelinpohjainen arkkitehtuuri erottaa laskennan pelilogiikasta. Tukee RVO-väistelyä. Toimii A*-algoritmilla

Taulukko 1. (jatkuu)

Ominaisuus/ Pelimoottori	Unity	Unreal Engine	Godot
Generatiivinen tekoäly ja Inferenssi	**Sentis & Muse:** Neuroverkkojen ajo päätelaitteella (Sentis) ja editoriin integroitu luonti.	**PCG Framework:** Sääntöpohjainen ja graafinen työkalu maailmanluontiin. NNI mahdollistaa ML-pohjaisen animaation	**GDExtension ja Avoimuus:** Ei sisäänrakennettua GenAI-ratkaisua. GDExtension-järjestelmä mahdollistaa ulkoisten koneoppimiskirjastojen ja neuroverkkojen tuomisen osaksi moottoria

Vertailun perusteella voidaan tunnistaa kolme erilaista kehitysfilosofiaa. Unity on omaksunut tuottavuusvetoisen strategian, jossa tekoäly pyritään demokratisoimaan. Tämä näkyy erityisesti Unityn AI Assistant -työkalussa, joka eroaa yleisistä kielimalleista kyvyllään ymmärtää projektin sisäisen kontekstin ja rakenteen (Unity Technologies, 2025c). Unreal Engine 5 keskittyy simulaation uskottavuuteen ja skaalautuvuuteen tarjoten raskaan sarjan työkaluja, kuten *MassEntity*-järjestelmän, jotka mahdollistavat kymmenien tuhansien agenttien autonomisen toiminnan laajoissa pelimaailmoissa (Epic Games, 2025f). Godot 4 edustaa arkkitehtuurista vapautta, jossa suorituskyky taataan palvelinpohjaisella rakenteella, mutta korkeamman tason logiikan toteutus jätetään tietoisesti kehittäjän tai yhteisön vastuulle ilman sitovia malleja (Godot Engine, 2020).

Merkittävin teknologinen murros on siirtymä kohti ajonaikaista tekoälyä, jossa deterministinen pelilogiikka ja stokastiset neuroverkkomallit toimivat rinnakkain. Vaikka molemmat markkinajohtajat tukevat tätä kehitystä, niiden soveltamiskohteet eroavat toisistaan. Unityn Sentis-järjestelmä tuo neuroverkkojen inferenssin osaksi yleistä pelimekaniikkaa ja päätöksentekoa (Unity Technologies, 2025e), kun taas Unreal Engine hyödyntää NNI-teknologiaa ensisijaisesti grafiikkaputken tehostamiseen, esimerkiksi reaaliaikaisen hahmoanimaation kautta (Epic Games, 2025b). Tämä kehitys viittaa siihen, että pelimoottori on muuttumassa passiivisesta suoritusympäristöstä aktiiviseksi inferenssialustaksi. Kuten luvussa 3.4.2 todettiin, tämä siirtymä muokkaa kehittäjän roolia, missä sääntöjen koodaamisen rinnalle nousee tarve hallita tekoälyä luonnollisella kielellä ja ymmärtää ennustavien mallien rajoitteita.

4 Yhteenveto ja johtopäätökset

Tämän kandidaatintutkielman tavoitteena oli tarkastella pelimoottoreita tietojärjestelmätieteen näkökulmasta ja selvittää, miten ne eroavat perinteisistä ohjelmistokehitysympäristöistä sekä miten tekoäly muuttaa pelinkehitysprosessia. Tutkielma toteutettiin kirjallisuuskatsauksena, jossa analysoitiin alan tutkimuskirjallisuutta ja kolmen merkittävän pelimoottorin, Unityn, Unreal Enginen ja Godotin teknistä dokumentaatiota.

4.1 Hybridimalli ja tekoälyintegraatio

Ensimmäinen tutkimuskysymys selvitti, miten pelimoottoripohjainen kehitys eroaa perinteisestä ohjelmistokehityksestä työkalujen ja abstraktiotason näkökulmasta. Tutkimus osoittaa, että pelikehitys on luonteeltaan hybridimalli, joka vaatii sekä koodikeskeistä osaamista että visuaalista sisällönhallintaa. Siinä missä perinteinen IDE-työskentely nojaa tekstimuotoiseen lähdekoodiin, pelimoottorit ovat nousseet korkean abstraktiotason alustoiksi, jotka piilottavat matalan tason teknisen toteutuksen valmiiden alijärjestelmien taakse. Tämä abstraktio on välttämätön, sillä peliprojekteissa jopa 70 prosenttia tiedostoista voi olla muuta kuin koodia. Pelimoottorit ratkaisevat "koodin ja taiteen välisen jaon” tarjoamalla visuaalisia työkaluja, jotka mahdollistavat pelilogiikan kehittämisen ilman perinteisen koodin kirjoittamista.

Toinen tutkimuskysymys tarkasteli tekoälytyökalujen ja proseduraalisen sisällöntuotannon vaikutusta pelikehitykseen. Tulokset osoittavat, että ala on parhaillaan murroksessa, jossa siirrytään deterministisistä säännöistä kohti todennäköisyyksiin perustuvaa generatiivista tekoälyä. Perinteinen proseduraalinen sisällöntuotanto on taannut kehittäjälle tarkan kontrollin lopputuloksesta, mutta uudet FMware-pohjaiset ratkaisut tuovat prosessiin stokastisuutta ja epävarmuutta. Tämä muuttaa kehittäjän roolia merkittävästi ensisijaisesti korkeamman abstraktiotason tehtävissä. Erityisesti sisällöntuotannossa kehittäjän rooli on siirtymässä teknisestä toteuttajasta kohti "kuraattoria" tai kehotetekniikan osaajaa, joka ohjaa generatiivista tekoälyä luonnollisella kielellä ja validoi sen tuottamaa sisältöä. On kuitenkin huomioitava pelikehityksen hybridimallin jatkuminen. Matalan tason toiminnot, kuten arkkitehtuurin suunnittelu, suorituskykyoptimointi ja muistinhallinta, edellyttävät yhä syvällistä koodikeskeistä erikoisosaamista. Täten kehittäjän roolimuutos kohdistuu ensisijaisesti sisällöntuotantoon ja visuaalisen logiikan luomiseen.

4.2 Teknologinen kaksijakoisuus ja laadunvarmistus

Tutkielma osoittaa, että pelikehitys on kokemassa merkittävän paradigman muutoksen. Pelimoottorit eivät enää toimi vain passiivisina suoritusympäristöinä, vaan ne integroivat neuroverkkoja yhä syvemmin suoraan ajonaikaiseen ympäristöön. Tämä kehitys laajentaa pelitekoälyn perinteistä määritelmää. Tekoäly ei enää rajoitu vain NPC-hahmojen logiikan ohjaamiseen, vaan kattaa myös reaaliaikaisen, tekoälyavusteisen sisällön generoinnin. Näin ala on siirtymässä deterministisestä koodauksesta kohti todennäköisyyksiin perustuvaa toimintamallia.

Tämä siirtymä luo teknologisen kaksijakoisuuden. Vaikka generatiivinen tekoäly tavoittelee nopeaa sisällön määrän kasvua ja luovaa variaatiota, pelimoottorit kehittävät samanaikaisesti uusia, deterministisiä arkkitehtuureja, kuten Unreal Enginen *MassEntity*-järjestelmä ja Godotin *Navigation Server*. Näiden arkkitehtuurien tarkoituksena on juuri ylläpitää suorituskykyä, vakautta ja ennustettavuutta suurten väkijoukkojen ja monimutkaisen pelilogiikan hallinnassa. Pelialan kehitys pyrkii siis tasapainottamaan tekoälyn tuoman sisällöntuotannon nopeuden ja perinteisen arkkitehtuurin takaaman deterministisen vakauden

Tämä kehitys luo kuitenkin uudenlaisen haasteen ohjelmistotuotannon laadunvarmistukselle. Perinteisessä ohjelmistokehityksessä virheet johtuvat loogisista ristiriidoista, jotka voidaan korjata koodia muokkaamalla. Generatiivisessa tekoälyssä virheet ilmenevät hallusinaatioina tai epätoivottuna varianssina, jota on vaikea hallita perinteisin testausmenetelmin. Tämä havainto on ristiriidassa peliteollisuuden perinteisen, tarkan kontrollin tavoittelun kanssa.

Toisaalta pelimoottorien standardoitu arkkitehtuuri tarjoaa tekoälylle otollisen toimintaympäristön, joka voi lieventää näitä laatuongelmia. Koska pelimoottorit toimivat ohjelmistokehyksinä ne tarjoavat selkeät rajat ja rakenteet, joiden puitteissa tekoäly operoi. Tämä mahdollistaa sen, että integroidut tekoälytyökalut voivat ymmärtää projektin syvällisen kontekstin paremmin kuin yleiskäyttöiset kielimallit. Tämä kontekstitietoisuus auttaa sitomaan generatiivisen tekoälyn tiukemmin pelinkehityksen käytännön tarpeisiin.

4.3 Tutkimuksen arviointi ja jatkotutkimustarpeet

Tutkielman luotettavuutta arvioitaessa on huomioitava, että kyseessä on kirjallisuuskatsaus. Työn aineisto koostuu vertaisarvioiduista tieteellisistä julkaisuista sekä pelimoottorien virallisesta teknisestä dokumentaatiosta. Koska peliteknologia ja erityisesti generatiivinen tekoäly kehittyy erittäin nopeasti, pelkkä tutkimustieto ei välttämättä pysy alan muutosten vauhdissa. Tässä työssä

tätä haastetta ratkaistiin yhdistämällä akateemista tutkimustietoa tuoreeseen pelimoottoridokumentaatioon, mikä mahdollisti sekä teoreettisen tarkastelun että ajantasaisen teknologiakatsauksen.

Tutkimuksen rajoitteena voidaan pitää sitä, ettei työssä toteutettu empiiristä kokeilua eri moottorien tekoälytyökaluilla. Siksi johtopäätökset työkalujen tehokkuudesta perustuvat dokumentoituihin ominaisuuksiin, eivät mitattuun suorituskykyyn. Lisäksi on huomioitava tarkasteltavan teknologian uutuus. Koska osa käsitellyistä tekoälytyökaluista, kuten generatiiviset avustajat, ovat tulleet markkinoille vasta viime vuosina, tutkimus pystyi arvioimaan vain niiden välittömiä vaikutuksia kehitystyöhön. Työkalujen pitkän aikavälin vaikutukset, kuten pelikoodin ylläpidettävyys ja teknisen velan kertyminen, jäävät siten tämän tutkimuksen ulkopuolelle. Samalla tekoälytyökalujen nopea päivitystahti tarkoittaa, että tutkielmassa esitetyt ominaisuudet voivat muuttua tai vanhentua nopeasti

Tämän tutkielman perusteella nousee esiin selkeä tarve tutkia generatiivisen tekoälyn vaikutusta peliohjelmistojen ylläpidettävyyteen ja tekniseen velkaan. Erityisesti tulisi selvittää, miten vastuu ohjelmiston elinkaaresta ja ylläpidosta määritellään pitkällä aikavälillä tilanteissa, joissa merkittävä osa pelikoodista tai sisällöstä on tekoälyn generoimaa. Toinen keskeinen jatkotutkimusaihe liittyy tekijänoikeuksiin ja eettisyyteen. Tulevaisuudessa on syytä tarkastella kriittisesti, millaisia vaikutuksia pelimoottoreihin integroiduilla, avoimella datalla koulutetuilla tekoälymalleilla on pelistudioiden immateriaalioikeuksien hallinnalle.

Lähteet

Agrahari, V., & Chimalakonda, S. (2021). What's Inside Unreal Engine? - A Curious Gaze! *Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, 1–5. https://doi.org/10.1145/3452383.3452404

Cai, H. (2015). Facilitating Information Management in Integrated Development Environments through Visual Interface Enhancements. *Reliability and Security - Companion 2015 IEEE International Conference on Software Quality*, 221–229. https://doi.org/10.1109/QRS-C.2015.46

Chen, X. (2025). Exploring GenAI-Driven Innovation in Game Development. *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 157–159. https://doi.org/10.1109/ICSE-Companion66252.2025.00046

Chueca, J., Trasobares, J. I., Domingo, Á., Arcega, L., Cetina, C., & Font, J. (2023). Comparing software product lines and Clone and Own for game software engineering under two paradigms: Model-driven development and code-driven development. *Journal of Systems and Software*, *205*, 111824. https://doi.org/10.1016/j.jss.2023.111824

Gasparic, M., Gurbanov, T., & Ricci, F. (2017). Context-aware integrated development environment command recommender systems. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 688–693. https://doi.org/10.1109/ASE.2017.8115679

Hassan, A. E., Lin, D., Rajbahadur, G. K., Gallaba, K., Cogo, F. R., Chen, B., Zhang, H., Thangarajah, K., Oliva, G., Lin, J. (Justina), Abdullah, W. M., & Jiang, Z. M. (Jack). (2024). Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 294–305. https://doi.org/10.1145/3663529.3663849

Hou, J. (Jove), Wang, L., Wang, G., Wang, H. J., & Yang, S. (2025). The Double-Edged Roles of Generative AI in the Creative Process: Experiments on Design Work. *Information Systems Research*. https://doi.org/10.1287/isre.2024.0937

Kasurinen, J. (2016). Games as Software: Similarities and Differences between the Implementation Projects. *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, 33–40. https://doi.org/10.1145/2983468.2983501

Lazaridis, L., Kollias, K.-F., Maraslidis, G., Michailidis, H., Papatsimouli, M., & Fragulis, G. F. (2022). Auto Generating Maps in a 2D Environment. Teoksessa X. Fang (Toim.), *HCI in Games* (s. 40–50). Springer International Publishing. https://doi.org/10.1007/978-3-031-05637-6_3

Martínez, Y., Cachero, C., & Meliá, S. (2014). Empirical study on the maintainability of Web applications: Model-driven Engineering vs Code-centric. *Empirical Software Engineering*, *19*(6), 1887–1920. https://doi.org/10.1007/s10664-013-9269-5

Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? *Proceedings of the 36th International Conference on Software Engineering*, 1–11. https://doi.org/10.1145/2568225.2568226

Pascarella, L., Palomba, F., Di Penta, M., & Bacchelli, A. (2018). How is video game development different from software development in open source? *Proceedings of the 15th International Conference on Mining Software Repositories*, 392–402. ICSE '18: 40th International Conference on Software Engineering. https://doi.org/10.1145/3196398.3196418

Politowski, C., Petrillo, F., Montandon, J. E., Valente, M. T., & Guéhéneuc, Y.-G. (2021). Are game engines software frameworks? A three-perspective study. *Journal of Systems and Software*, *171*, 110846. https://doi.org/10.1016/j.jss.2020.110846

Ricca, F., Torchiano, M., Leotta, M., Tiso, A., Guerrini, G., & Reggio, G. (2018). On the impact of state-based model-driven development on maintainability: A family of experiments using UniMod. *Empirical Software Engineering*, *23*(3), 1743–1790. https://doi.org/10.1007/s10664-017-9563-8

Risi, S., & Preuss, M. (2020). From Chess and Atari to StarCraft and Beyond: How Game AI is Driving the World of AI. *KI - Künstliche Intelligenz*, *34*(1), 7–17. https://doi.org/10.1007/s13218-020-00647-w

Rozo-Torres, A., & Sarmiento, W. J. (2024). Prompt Engineering, An Alternative for Video Game Development? Teoksessa P. H. Ruiz, V. Agredo-Delgado, & A. Mon (Toim.), *Human-Computer Interaction* (s. 242–256). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-57982-0_19

Salama, R., & Elsayed, M. (2021). A live comparison between Unity and Unreal game engines. *Global Journal of Information Technology: Emerging Technologies*, *11*, 01–07. https://doi.org/10.18844/gjit.v11i1.5288

Torres-Ferreyros, C. M., Festini-Wendorff, M. A., & Shiguihara-Juárez, P. N. (2016). Developing a videogame using unreal engine based on a four stages methodology. *2016 IEEE ANDESCON*, 1–4. https://doi.org/10.1109/ANDESCON.2016.7836249

Ullmann, G. C., Guéhéneuc, Y.-G., Petrillo, F., Anquetil, N., & Politowski, C. (2023). An Exploratory Approach for Game Engine Architecture Recovery. *2023 IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS)*, 8–15. https://doi.org/10.1109/GAS59301.2023.00009

Wang, A. I., & Nordmark, N. (2015). Software Architectures and the Creative Processes in Game Development. Teoksessa K. Chorianopoulos, M. Divitini, J. Baalsrud Hauge, L. Jaccheri, & R. Malaka (Toim.), *Entertainment Computing—ICEC 2015* (s. 272–285). Springer International Publishing. https://doi.org/10.1007/978-3-319-24589-8_21

Yannakakis, G. N., & Togelius, J. (2018). *Artificial Intelligence and Games*. Springer International Publishing. https://doi.org/10.1007/978-3-319-63519-4

Pelimoottorianalyysin lähteet

Epic Games. (2025a). Behavior Tree in Unreal Engine - Overview. Unreal Engine 5.5 Documentation. Viitattu 2.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---overview

Epic Games. (2025b). ML Deformer Sample. Fab / Unreal Engine Resources. Viitattu 14.12.2025. Saatavilla: https://www.fab.com/listings/1b853072-1125-4602-955d-4a232b918475

Epic Games. (2025c). Neural Network Inference (NNI). Unreal Engine 5.5 Documentation. Viitattu 14.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/neural-network-inference-in-unreal-engine

Epic Games. (2025d). Overview of State Tree in Unreal Engine. Unreal Engine 5.5 Documentation. Viitattu 12.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-state-tree-in-unreal-engine

Epic Games. (2025e). Procedural Content Generation Framework. Unreal Engine 5.5 Documentation. Viitattu 12.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-framework-in-unreal-engine

Epic Games. (2025f). Mass Entity in Unreal Engine. Unreal Engine 5.5 Documentation. Viitattu 12.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/mass-entity-in-unreal-engine

Epic Games. (2025g). Zone Graph in Unreal Engine. Unreal Engine 5.5 Documentation. Viitattu 12.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/zone-graph-in-unreal-engine

Epic Games. (2025h). Mass Avoidance in Unreal Engine. Unreal Engine 5.5 Documentation. Viitattu 12.12.2025. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/mass-avoidance-in-unreal-engine

Godot Engine. (2020, 19. helmikuuta). Navigation Server for Godot 4.0. Godot Engine Blog. Viitattu 12.12.2025 osoitteesta https://godotengine.org/article/navigation-server-godot-4-0/

Godot Engine. (2025a). AnimationTree. Godot Engine 4.3 Documentation. Viitattu 2.12.2025. Saatavilla: https://docs.godotengine.org/en/stable/classes/class_animationtree.html

Godot Engine. (2025b). GDExtension. Godot Engine 4.3 Documentation. Viitattu 12.12.2025. Saatavilla:

https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html

Godot Engine. (2025c). NavigationRegion3D. Godot Engine 4.3 Documentation. Viitattu 2.12.2025 osoitteesta https://docs.godotengine.org/en/stable/classes/class_navigationregion3d.html

Godot Engine. (2025d). NavigationServer3D. Godot Engine 4.3 Documentation. Viitattu 12.12.2025. Saatavilla: https://docs.godotengine.org/en/stable/classes/class_navigationserver3d.html

Godot Engine. (2025e). Using NavigationAgents. Godot Engine 4.3 Documentation. Viitattu 2.12.2025. Saatavilla: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationagents.html

Godot Engine. (2025f). Navigation Runtime Baking. Godot Engine 4.3 Documentation. Viitattu 2.12.2025. Saatavilla: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationmeshes.html#runtime-baking

Unity Technologies. (2025a). NavMesh Surface. AI Navigation Package 1.1 Documentation. Viitattu 2.12.2025. Saatavilla: https://docs.unity3d.com/Packages/com.unity.ai.navigation@1.1/manual/NavMeshSurface.html

Unity Technologies. (2025b). NavMesh Link. AI Navigation Package 1.1 Documentation. Viitattu 2.12.2025. Saatavilla: https://docs.unity3d.com/Packages/com.unity.ai.navigation@1.1/manual/NavMeshLink.html

Unity Technologies. (2025c). Unity AI. Unity 6 Documentation. Viitattu 14.12.2025. Saatavilla: https://docs.unity3d.com/6000.2/Documentation/Manual/unity-ai.html

Unity Technologies. (2025d). Unity Behavior. Unity 6 Documentation. Viitattu 12.12.2025. Saatavilla: https://docs.unity3d.com/Packages/com.unity.behavior@latest

Unity Technologies. (2025e). Unity Sentis. Unity Documentation. Viitattu 14.12.2025. Saatavilla: https://docs.unity3d.com/Packages/com.unity.sentis@latest

Liitteet

Hyödynsin OpenAI:n Chatgpt -kielimallia aiheen alkuvaiheen ideoinnissa ja tutkimuskysymysten muotoilun tukena. Käytin työkalua apuna hahmotellessani työn rakennetta ja sisällysluetteloa.

Olen käyttänyt Googlen Gemini -tekoälyä tekstin oikolukemiseen ja lauserakenteiden sujuvoittamiseen. Olen syöttänyt työkaluun kirjoittamiani tekstikappaleita ja pyytänyt parannusehdotuksia kieliasuun.

Käytin Googlen Gemini Deep Research -työkalua kartoittamaan pelimoottoreiden teknisiä dokumentteja, mutta en ole käyttänyt tekoälyä suorana viitattavana tiedonlähteenä, vaan olen tarkistanut kaikki työkalun ehdottamat lähteet alkuperäisjulkaisuista ennen niiden sisällyttämistä työhön.