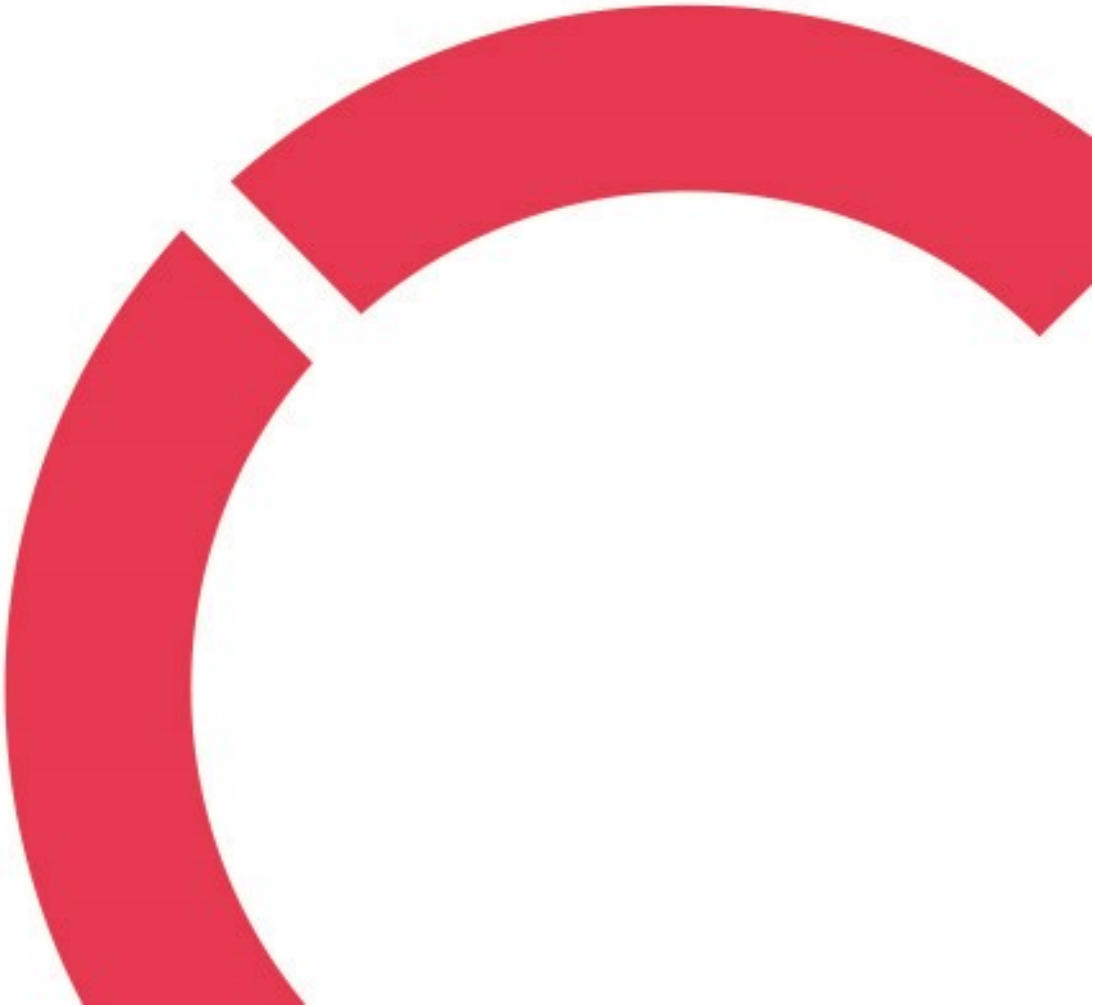
Joonas Hakatie

TEKOÄLY GODOT-PELIMOOTTORILLA

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikka
Kesäkuu 2025



TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Kesäkuu 2025	**Tekijä/tekijät** Joonas Hakatie
Koulutus Tieto-ja viestintätekniikka, Insinööri		☒ AMK ☐ YAMK
Työn nimi TEKOÄLY GODOT-PELIMOOTTORILLA		
Työn ohjaaja Sari Lipsanen		**Sivumäärä** 51
Työelämäohjaaja		

Opinnäytetyön aiheena oli tekoälyn hyödyntäminen Godot-pelimoottorissa. Tavoitteena oli tutkia, miten tekoälyä ja siihen liittyviä algoritmeja voidaan soveltaa pelikehityksessä, erityisesti Godot-pelimoottorin tarjoamassa ympäristössä. Työn osana kehitettiin demonstraatiosovellus, jossa tarkasteltiin, kuinka pelihahmon ohjaaminen onnistuu hyötypohjaista tekoälytekniikkaa hyödyntäen.

Opinnäytetyön teoriaosuudessa käsiteltiin aluksi, mitä tekoäly on ja mitä sillä tarkoitetaan pelikehityksen kontekstissa. Tämän jälkeen tarkasteltiin yleisimpiä tekoälyalgoritmeja sekä niiden soveltuvuutta peleihin. Lopuksi perehdyttiin Godot-pelimoottorin ominaisuuksiin ja sen tarjoamiin mahdollisuuksiin tekoälyn toteuttamisessa.

Käytännön osuudessa suunniteltiin ja toteutettiin hyötypohjainen tekoälyratkaisu Godot-pelimoottorilla. Ratkaisun toimintaa testattiin demopelissä, jossa tekoäly ohjasi pelihahmon käyttäytymistä valittujen hyötyarvojen perusteella.

Työn aikana havaittiin, että hyötypohjainen tekoäly soveltuu hyvin Godot-pelimoottoriin ja toimii odotusten mukaisesti. Toteutus osoitti, että myös ilman valmiita tekoälytyökaluja on mahdollista rakentaa toimiva järjestelmä, joka tarjoaa monipuolista ja reaktiivista pelihahmon käyttäytymistä.

Asiasanat
Tekoäly, Godot, pelimoottori

ABSTRACT

Centria University of Applied Sciences	Date June 2025	Author Joonas Hakatie
Degree programme Information and Communication Technology, Engineer		
Name of thesis ARTIFICIAL INTELLIGENCE WITH GODOT GAME ENGINE		
Centria supervisor Sari Lipsanen		**Pages** 52
Instructor representing commissioning institution or company		

The topic of this thesis was the utilization of artificial intelligence in the Godot game engine. The aim was to explore how artificial intelligence, and related algorithms can be applied in game development, particularly in the environment provided by the Godot game engine. As part of the thesis, a demonstration application was developed to examine how character control can be achieved using utility-based AI techniques.

The theoretical section of the thesis covered what artificial intelligence is and what it means in the context of game development. This was followed by an exploration of the most common AI algorithms and their suitability for games. Finally, the properties of the Godot game engine and the possibilities it offers for implementing AI were examined.

In the practical section, a utility-based AI solution was designed and implemented in the Godot game engine. The solution was tested in a demo game, where the AI controlled the character's behavior based on selected utility values.

During the work, it was observed that utility-based AI works well within the Godot game engine and functions as expected. The implementation showed that it is possible to build a functional system that provides diverse and reactive character behavior, even without using pre-built AI tools.

Key words
Artificial intelligence, Godot, game engine

KÄSITTEIDEN MÄÄRITTELY

Pelimoottori

Pelien luontiin tarkoitettu ohjelma. Pelimoottori on ohjelmistoalusta, joka kokoaa yhteen erilaiset työkalut ja kehykset pelin rakentamista varten. Se tarjoaa valmiita ratkaisuja esimerkiksi grafiikan, fysiikan, äänen, tekoälyn ja pelilogiikan käsittelyyn, minkä ansiosta kehittäjät voivat keskittyä enemmän pelin ideointiin ja sisällön luomiseen. Pelimoottorit nopeuttavat ja helpottavat pelinkehitystä merkittävästi, ja useimmat niistä mahdollistavat myös toimintojen laajentamisen omien ratkaisujen avulla. Suosittuja pelimoottoreita ovat esimerkiksi Unity, Unreal Engine ja Godot.

NPC

Pelihahmoa, jota pelaaja ei itse ohjaa, kutsutaan NPC:ksi (engl. *Non Player Character*). Tällaista hahmoa ohjataan peliin kehitetyllä ohjelmoinnilla ja tekoälyllä. NPC:t voivat toimia pelissä erilaisissa rooleissa, kuten esimerkiksi vastuksina tai liittolaisina. Toinen samankaltainen sana on *botti*, jonka voidaan ajatella olevan mahdollisesti tutumpi ja laajempi sana, mutta sillä voidaan tarkoittaa myös muita asioita. Tässä työssä käytetään selvyyden vuoksi lyhennettä NPC.

Älykäs peliagentti

Pelissä toimiva hahmo tai järjestelmä, joka käyttää tekoälyä tehdäkseen itsenäisiä päätöksiä ja reagoidakseen ympäristöönsä. Tällainen agentti voi olla esimerkiksi pelihahmo, vihollinen tai sillä voidaan viitata pelissä järjestelmään, joka reagoi pelaajan toimiin, oppii kokemuksistaan ja mukautuu pelin muutoksiin.

Skripti

Kooditiedosto, joilla koodataan toiminnallisuuksia Godotissa.

Noodi

Godotissa noodi (engl. *Node*), jota voidaan kutsua myös *solmuksi*, on erittäin tärkeä käsite. Teknisesti noodi on luokka, joka toimii objektin pohjana ja tarjoaa perustoiminnallisuuden, kuten sijainnin, kierron ja skaalauksen (transformaatio), sekä mahdollisuuden lisätä lapsisolmuja ja yhdistää skriptejä. Noodi on perusrakennuspalikka, joka mahdollistaa pelin eri osien, kuten grafiikan, fysiikan ja logiikan, järjestämisen ja hallinnan. Tässä työssä termiä noodi käytetään viitattaessa Godot-pelimoottorin noodijärjestelmään. Muissa yhteyksissä, kuten algoritmien solmukohtien kuvauksessa, käytetään termiä solmu.

Skene

Skene (engl. *Scene*) on Godot-pelimoottorin perusrakenne, jota käytetään pelin sisällön luomiseen ja järjestämiseen loogisiksi kokonaisuuksiksi.

Pelialalla usein työkielenä on englanti, ja termeistä saatetaan käyttää yleisesti niiden englanninkielisiä nimiä. Tässä työssä on pyritty löytämään termeille niille suomenkielisesti sopivat vastineet ja käyttämään niitä johdonmukaisesti.

1 JOHDANTO

Tekoäly (engl. *Artificial Intelligence, AI*) on olennainen osa nykyajan digitaalista pelialaa, ja sen roolin ja käyttömahdollisuuksien pelikehityksessä voidaan nähdä olevan edelleen laajentumassa. Tekoäly voidaan määritellä tietokoneohjelman kykynä suorittaa älykkäinä pidettäviä toimintoja, jotka muistuttavat elävien olentojen ajattelua ja päätöksentekoa (Millington 2019, 4). Pelimaailmassa tekoäly ilmenee monin eri tavoin, kuten esimerkiksi NPC-hahmoina (*Non-Player Characters*), jotka voivat toimia vihollisina, liittolaisina tai itsenäisesti toimivina pelihahmoina. Moninpeleissä (engl. *Multiplayer*) tekoäly voi korvata ihmispelaajat ”botteina”, joten peli säilyy täysipainoisena myös ilman oikeiden pelaajien läsnäoloa. Tekoälyn avulla voidaan keskeisesti parantaa pelin dynamiikkaa ja rikastuttaa pelaajakokemusta. Tekoäly mahdollistaa muun muassa dynaamisten ja reaktiivisten pelimaailmojen luomisen, älykkään vastarinnan ja realistisen NPC-käyttäytymisen. Lisäksi tekoälyn avulla voi ohjata esimerkiksi pelihahmojen navigointia, mukauttaa vaikeustasoa, luoda tarinaa ja vuorovaikutusta sekä lisätä pelin henkilökohtaisuutta tai personointia. Tekoälyn rooli vaihtelee pelin tyypin ja toiminnallisten vaatimusten mukaan, ja sen käyttö tulee todennäköisesti tulevaisuudessa yhä monipuolistumaan ja tarjoamaan uusia mahdollisuuksia pelien kehittämiseen.

Tekoäly on kehittynyt merkittävästi viime vuosikymmeninä, ja nykyisin peliteollisuus hyödyntää laajasti monenlaisia algoritmeja ja tekniikoita: perinteisiä reitinhakualgoritmeja, päätöksenteko- ja hakualgoritmeja, käyttäytymismalleja, tilannekohtaisia järjestelmiä ja uudemman sukupolven syväoppimis- ja neuroverkkoja. Pelit tarjoavat myös tekoälyn tutkimukselle ainutlaatuisen kentän, sillä ne luovat monimutkaisia päätöksentekotilanteita ja laajoja hakutiloja, joissa tekoälyn on pystyttävä tekemään älykkäitä valintoja rajallisissa ja usein muuttuvissa olosuhteissa. Päätöksenteko voi monesti olla myös reaaliaikaista. Lisäksi pelit synnyttävät mielenkiintoisen tutkimusnäkökulman tekoälyyn, sillä ne tarjoavat mahdollisuuden koettavissa olevalle luovuudelle ja ilmaisulle. Näiden haasteiden ja mahdollisuuksien takia pelit ovat houkutteleva kenttä tekoälyn tutkimukselle ja pelit tarjoavat mahdollisuuksia algoritmien ja järjestelmien testaamiseen ja parantamiseen. (Yannakakis & Togelius 2018, 4.)

Tässä opinnäytetyössä tarkastellaan tekoälyn käyttöä erityisesti Godot-pelimoottorissa. Godot on avoimen lähdekoodin pelimoottori, joka tarjoaa sekä monipuoliset työkalut pelikehittäjille että laajat mahdollisuudet tekoälyn kehittämiseen. Eri pelimoottorit tarjoavat vaihtelevasti valmiita työkaluja tekoälyn kehittämiseen, ja näiden työkalujen valikoimaa voidaan tarvittaessa laajentaa. Valmiiden työkalu-

jen käytön lisäksi tekoälytoimintoja voidaan kehittää koodiskripteillä. Tekoälytoimintojen kehittäminen koodiskriptien avulla mahdollistaa samanlaisten algoritmien ja kehityslähestymistapojen soveltamisen riippumatta pelimoottorista. Työssä edetään niin, että ensin perehdytään tekoälyn perusteisiin ja sen soveltamiseen peleissä keskittyen erityisesti päätöksentekoalgoritmeihin ja niiden soveltamiseen pelien kehityksessä. Tämän jälkeen perehdytään Godot-pelimoottoriin sekä sen tarjoamiin mahdollisuuksiin tekoälyn kehittämisessä. Opinnäytetyön lopuksi toteutetaan demosovellus, jossa testataan tekoälyn käyttötapoja ja algoritmeja käytännössä Godot-pelimoottorissa.

Tutkimuskysymykseni ovat:

Miten tekoälyalgoritmeja voidaan käyttää peleissä?
Kuinka Godot-pelimoottorissa voidaan kehittää tekoälyä peleihin?
Mitä tekijöitä on huomioitava, kun Godotissa kehitetään tekoälyä peliin?

2 TEKOÄLYSTÄ YLEISESTI

Tekoälytutkimus keskittyy älykkäiden järjestelmien ja entiteettien kehittämiseen sekä niiden toiminnan ymmärtämiseen. Tekoäly on nuori tieteenala, joka sai akateemisen perustan 1950-luvulla, mutta on sen jälkeen kehittynyt nopeasti ja laajentunut useille eri osa-alueille. Viime vuosikymmeninä tekoäly on noussut merkittäväksi tutkimusalueeksi ja saanut yhä enemmän huomiota käytännön sovelluksissa. Tekoäly kattaa laajan kentän, johon kuuluvat muun muassa koneoppiminen, syväoppiminen, luonnollisen kielen käsittely, tietokonenäkö ja robotiikka. (Russell & Norvig 2022.) Tekoälyyn liittyvien teknisten haasteiden ohella siihen liittyy myös eettisiä ja yhteiskunnallisia kysymyksiä, kuten yksityisyyden-suoja, vaikutukset työmarkkinoihin ja päätöksenteon läpinäkyvyys. Näihin haasteisiin etsitään ratkaisuja samalla, kun tekoälyjärjestelmiä kehitetään monimutkaisemmiksi ja tehokkaammiksi.

Tekoälyn perusidea on, että se voi automaattisesti suorittaa tehtäviä, jotka perinteisesti ovat vaatineet inhimillistä älykkyyttä, kuten ongelmanratkaisua, päätöksentekoa, kielen ymmärtämistä ja luovuutta. Tekoälyjärjestelmät voivat esimerkiksi tunnistaa kuvissa esineitä, kääntää tekstejä, pelata shakkia tai luoda taidetta. Käyttö on laajentunut monille aloille, kuten terveydenhuoltoon, rahoitukseen, liikenteeseen, viihteeseen ja asiakaspalveluun. Tekoälyä hyödynnetään muun muassa lääketieteellisessä diagnostiikassa, taloudellisessa ennustamisessa, itseajavissa autoissa, viihteen tuottamisessa ja älykkäissä avustajissa. Koska tekoäly mahdollistaa älykkyyttä vaativien tehtävien automaation, sen sovelluksien voidaan nähdä ulottuvan lähes kaikille älykästä toimintaa vaativille alueille.

Ian Millington (2019, 6-7) kuvaa kirjassaan *Artificial Intelligence for Games* tekoälyn kehitystä jakamalla sen kolmeen eri aikakauteen: varhaisiin aikoihin (engl. *early days*), symboliseen aikakauteen (engl. *symbolic era*) ja biologisia järjestelmiä mallintavaan aikakauteen (engl. *natural era*). Alkuvaiheessa, ennen tietokoneiden aikakautta, ihmiset pohtivat kysymystä voiko elottomasta objektista tehdä elollisen. Tämä pohdinta oli pitkälti teoreettista ja filosofista. Tekoälyn alkuvaiheessa keskiössä olivat enemmänkin kysymykset tietoisuuden ja älykkyyden luonteesta kuin teknologinen kehitys. Symbolisen vaiheen voidaan katsoa alkaneen 1950-luvun lopulla, jolloin kiinnostus kohdistui erityisesti symbolisiin järjestelmiin. Näissä järjestelmissä algoritmeista voidaan erottaa kaksi selkeää komponenttia: data, joka voi olla sanoja, numeroita, lauseita, kuvia tai muuta informaatiota, sekä algoritmi, joka manipuloi tätä dataa ja tuottaa sen avulla uutta tietoa tai ratkaisee ongelmia. Tällaisia lähestymistapoja ovat muun muassa asiantuntijajärjestelmät (engl. *expert systems*), blackboard-arkkitehtuurit, polunetsintä (engl. *pathfinding*) ja tilakoneet (engl. *state machines*). Usein näissä algoritmeissa hyödynnetään

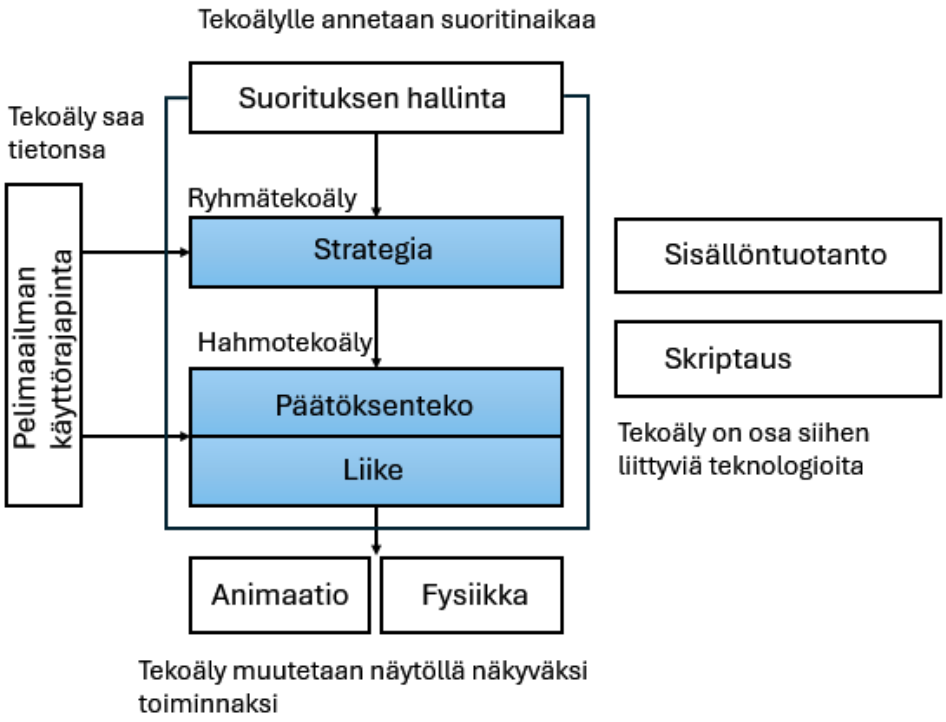
hakualgoritmeja, joissa etsitään parasta vaihtoehtoa optimaalisen ratkaisun löytämiseksi. Kuten Millington toteaa, symbolinen lähestymistapa onkin yhä laajasti käytössä esimerkiksi pelien tekoälyssä. (Millington 2019, 6–7.)

1980- ja 1990-luvuilla symboliset järjestelmät alkoivat osoittaa rajoituksiaan. Esimerkiksi puheen ymmärtämisessä, kuvantunnistuksessa ja luonnollisen kielen käsittelyssä kävi ilmi, että symboliset lähestymistavat eivät kyenneet käsittelemään monilla arkipäivän alueilla esiintyviä monimutkaisempia ja vähemmän strukturoituja tietoja. Tämän seurauksena kehitys suuntautui luonnollisempiin toimintoihin perustuviin tekniikoihin, kuten neuroverkkoihin, geneettisiin algoritmeihin ja koneoppimiseen. 2000-luvulta alkaen tekoälyssä tehtiin merkittäviä edistysaskelia. Tämän taustalla olivat muun muassa suorittimien laskentatehon merkittävä kasvu, käytettävissä olevan datan määrän lisääntyminen sekä tekniikoiden ja algoritmien kehittyminen. Näiden edistysten avulla pystyttiin ratkaisemaan ongelmia, jotka liittyivät muun muassa suurten tietomassojen käsittelyyn, kuvien ja videoiden tunnistamiseen, ennustamiseen ja kompleksisten ilmiöiden mallintamiseen. Erityisesti syvät neuroverkot (engl. *deep learning*) muodostivat perustan monille nykyisille tekoälysovelluksille, kuten itseoppiville järjestelmille, jotka pystyvät parantamaan suoritustaan kokemuksen kautta ilman suoraa ohjelmointia. Näiden uusien lähestymistapojen avulla tekoäly pystyi ratkaisemaan monimutkaisempia ja dynaamisempia ongelmia, joita symboliset järjestelmät eivät olleet kyenneet käsittelemään. Tämä avasi mahdollisuuksia täysin uudenlaisten sovellusten kehittämiseen eri aloilla. Millington kuitenkin toteaa, että on huomioitava, ettei jaottelu ole niin selkeä, sillä symbolisia algoritmeja käytetään edelleen osana nykyaikaisempia tekniikoita. (Millington 2019, 6–8.)

3 TEKOÄLYN KÄYTTÖ PELEISSÄ

Koska tekoäly on laaja käsite ja sen käyttö peleissä sisältää omia erityispiirteitään, on tärkeää selkeyttää aihealueeseen liittyviä käsitteitä ja rajauksia. Brian Schwab (2009, 2) määrittelee tekoälyn käytön peleissä näin: ”Pelin tekoäly on erityisesti peliin kirjoitettua koodia, joka saa tietokoneella ohjatut elementit näyttämään siltä, että ne tekevät älykkäitä päätöksiä, kun pelissä on useita vaihtoehtoja tiettyyn tilanteeseen. Tämä johtaa käyttäytymiseen, joka on oleellista, tehokasta ja hyödyllistä.” Schwab korostaa tässä määritelmässään sitä pelien tekoälylle ominaista piirrettä, että tekoälyn kehittämisessä painotetaan usein enemmän lopputulosta ja käyttäytymistä, kuin sitä miten lopputulokseen on päädytty. Usein riittää, että lopputulos näyttää toivotulta ja se koetaan halutun kaltaiseksi, sillä pelit on suunniteltu ensisijaisesti koettavaksi eikä pelaajan välttämättä tarvitse tietää, mitä pelin taustalla tapahtuu. (Schwab 2009, 2.)

Tässä työssä tarkastellaan tekoälyn käyttöä peleissä – eli tekniikoita, joiden avulla voidaan kehittää peleissä toimintoja, joiden perusteella voidaan tehdä älykkäitä valintoja. Näiden toimintojen luominen pohjautuu erilaisten algoritmien käyttöön. Pelimoottoreihin on usein kehitetty valmiiksi (tai niihin voidaan ladata lisäosina) näihin algoritmeihin perustuvia työkaluja. Peleissä tekoälytoimintoja edustavaa koodin osaa kutsutaan usein älykkääksi peliagentiksi. Se saa tietoa pelimaailmasta ja tekee sen pohjalta itsenäisiä päätelmiä toimintansa tueksi. Nykyaikaista pelien tekoälyä ei voi pitää puhtaasti reaktiivisena, toisin kuin esimerkiksi pelien alkuaikojen jotain tiettyä kaavaa tai johonkin ärsykkeeseen aina samalla tavalla reagoivaa tekoälyä. Tällaisia järjestelmiä ei usein pidetä tekoälynä, koska ne eivät sisällä älykästä päätöksentekoa, vaan toimivat ennalta määrätyn logiikan mukaan. Uudemmat tekoälytoiminnot peleissä käyttävät kehittyneempiä algoritmeja tai muita älykkyyttä simuloivia tekniikoita, joiden avulla ne voivat mukautua peliympäristön muutoksiin ja tehdä itsenäisiä päätöksiä. Näiden avulla tekoäly voi analysoida pelimaailman muuttuvia olosuhteita ja sopeutua niihin dynaamisesti. Sen tekemät itsenäiset päätökset voivat perustua esimerkiksi pelin muuttuviin tapahtumiin, aiempaan kokemukseen, pelaajan toimintaan tai vaikkapa pelin sisäisiin sääntöihin. Tällainen älykäs käyttäytyminen mahdollistaa sen, että peliagentit voivat mukauttaa strategioitaan, luoda uudenlaisia haasteita pelaajalle tai vaikka ennakoida tulevia tapahtumia. Tämä lisää pelien syvyyttä ja tarjoaa pelaajille entistä immersiivisemmän ja henkilökohtaisemman pelikokemuksen. (Schwab 2009, 4.)



KUVIO 1. Mallinnus pelin tekoälystä. (mukaillen Millington 2019, 10)

Pohdittaessa mikä osa peleissä on tekoälyä, on tärkeää huomioida, että tekoäly on kytköksissä moniin muihin pelin toimintoihin, joita voidaan pitää tekoälyn tukijärjestelminä. Tällaisia toimintoja voivat olla esimerkiksi tekoälylle tuleva tieto pelimaailmasta, animaatiot, pelattavuus, fysiikkasimulaatiot tai muut pelin osa-alueet, joita tekoäly voi ohjata. Tässä tulevat esiin haasteet tekoälyn rajojen määrittämisessä, sillä tekoälyn rooli pelissä ei ole aina selkeästi rajattu. Kuviossa 1 kuvataan, kuinka tekoälyn rooli peleissä voidaan jakaa usein kolmeen keskeiseen tehtävään: päätöksentekoon, strategian luomiseen ja liikkumiseen. Tekoäly on myös usein tiiviisti yhteydessä muihin pelin toimintoihin, kuten animaatioihin ja fysiikkamallinnukseen. Vaikka nämä tukijärjestelmät eivät itse sinänsä ole tekoälyä, ne vaikuttavat merkittävästi siihen, kuinka tekoäly näyttäytyy pelissä. Varsinaisena tekoälynä pidetään kuitenkin algoritmeja ja koodin osia, jotka tekevät älykkäitä valintoja silloin, kun pelissä on useita mahdollisia vaihtoehtoja ja suuntia. (Schwab 2009, 6–7.)

Puhuttaessa tekoälystä peleissä tehdään usein erottelu akateemisen tekoälyn tutkimuksenkehittämisen ja pelien tekoälyn välillä. Tämä erottelu on tarpeellinen, sillä näiden kahden lähestymistavan tavoitteet, rajaukset ja soveltamisalueet eroavat merkittävästi toisistaan. Akateemisessa suuntauksessa pyritään kehittämään ratkaisuja, jotka voivat soveltua moniin erilaisiin ympäristöihin ja sovelluksiin. Tässä kontekstissa tekoälyalgoritmit ja -mallit suunnitellaan käsittelemään monimutkaisempia ongelmia,

jotka vaativat suuria laskennallisia resursseja ja tarkkuutta. Pelien tekoäly puolestaan ei tarvitse tällaista laajaa yleistettävyyttä tai valtavia laskennallisia resursseja, vaan sen pääasiallisena tavoitteena on luoda pelaajalle uskottava, tasapainoinen ja mielenkiintoinen pelikokemus. Tieteen kentässä pelien tekoäly voidaan nähdä yhtenä tekoälyn tieteenalan alalajina, jolla on omat erityispiirteensä. (Millington 2019, 5–8.)

Peleissä tekoälyn tehtävänä on jäljitellä älykästä käyttäytymistä ja reagoida pelikokemukseen sopivalla tavalla. Siinä kontekstissa tekoäly ei pyri ratkaisemaan suuria teoreettisia ongelmia, vaan sen fokus on luoda dynaamisia ja mukautuvia peliympäristöjä, joissa esimerkiksi NPC-hahmot käyttäytyvät uskottavasti ja viihdyttävästi vuorovaikutuksessa pelaajan kanssa. Tärkeää on myös suorituskyvyn optimointi: pelin tekoälyn on toimittava sujuvasti, jotta pelaajan kokemus ei häiriinny, ja sen on oltava tarpeeksi nopea ja reagoitava ajallaan pelaajan toimiin, jotta peli pysyy nautittavana. Pelien tekoälyssä ei siis tarvitse välttämättä ratkaista monimutkaisia ongelmia tai käsitellä suuria tietomääriä, kuten akateemisessa tekoälyssä. Sen sijaan tekoäly toimii välineenä, joka palvelee pelin suunnittelua ja parantaa käyttäjäkokemusta. Esimerkiksi vihollishahmojen käyttäytyminen pelissä voi perustua yksinkertaisiin sääntöihin tai ennalta määrättyihin kaavoihin, mutta niiden on silti näytettävä ja tunnuttava pelaajasta haastavilta ja kiinnostavilta. Pelien tekoäly on siis erityisesti riippuvainen pelisuunnittelun tarpeista ja keskittyy pelaajakokemuksen parantamiseen. Pelien tekoälyssä on usein myös tärkeää löytää tasapaino: liian vahva tekoäly voi tehdä pelistä niin vaikean, että käyttäjä lopettaa pelaamisen, kun taas epäuskottava tai virheellinen käyttäytyminen voi saada pelin näyttämään epäkiinnostavalta. Tavoitteena on tarjota sopiva haaste ja uskottava kokemus. (Roberts 2023, 3.)

Ian Millington jakaa kirjassaan *Artificial Intelligence for Games* tekoälyn käytön kolmeen selkeään kokonaisuuteen: strategia, päätöksenteko ja liike. Strategisella tarkoitetaan tekoälyn kykyä luoda pidempiaikaisia strategioita hetkeen reagoimisen sijaan, sopeutumista peliympäristöön ja muuntautumista pelin tapahtumiin. Millington viittaa esimerkiksi tilanteeseen, jossa ohjataan useita pelihahmoja kerralla ja näiden tulee toimia strategisesti yhdessä, tai siihen että shakkipelissä tarvitaan vain strategista osaamista pelien siirtojen suunnittelussa. Päätöksenteolla tarkoitetaan tekoälyn kykyä valita seuraava toiminto. Tällaisia voisivat olla NPC-hahmolla vaikkapa vartiointi, paikallaan oleminen tai hyökkääminen. Päätöksentekojärjestelmä määrittää, mikä näistä toiminnoista on sopivin kussakin tilanteessa. Liikkumisella taas tarkoitetaan sellaisten algoritmien käyttöä, joiden avulla pelaajaa liikutetaan pelimaailmassa. Tällöin voidaan puhua erilaisista reitinhakualgoritmeista, joilla voidaan laskea optimaalisia reittejä hahmon liikuttelemiseen pelimaailmassa. Liikuteltavan hahmon on myös kyettävä

ottamaan huomioon edessä olevat objektit, pinnanmuodot tai muut esteet. On syytä huomioida myös se, että joskus hahmon liikkeet voidaan toteuttaa puhtaasti animaatioillakin. (Millington 2019, 5–8.)

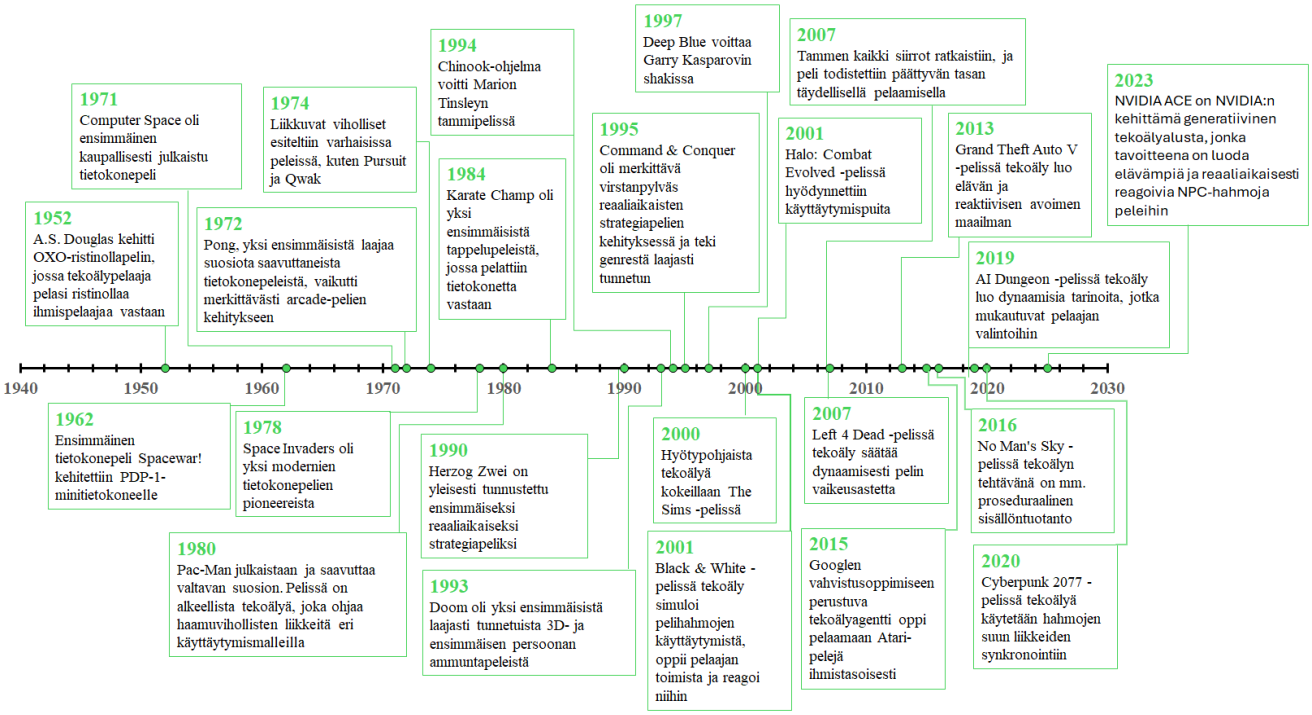
Tekoälyn kehittämisessä peleihin ei ole aina yhtä oikeaa tapaa toimia, sillä jokainen peli on ainutlaatuinen. Tekoälyn kehittäjän on osattava arvioida, millä tekniikoilla haluttu peli-idea voidaan saada toteutettua, ja usein ratkaisut ovat hyvin yksilöllisiä. Ei ole olemassa yksinkertaisesti ”parasta” tekniikkaa, joka toimisi kaikissa peleissä – eikä peleissä ole aina toivottavaa käyttää monimutkaisimpia tekniikoita. Millington toteaa *Artificial Intelligence for Games* -teoksessaan, että pelissä yksinkertainen tekniikka voi näyttää älykkäämmältä kuin se todellisuudessa on, koska peleissä usein riittää, miltä asiat näyttävät pelaajalle. (Millington 2019, 6–8.)

Tekoälytoimintojen kehittämisessä voi kohdata rajoituksia, kuten esimerkiksi sen, että neuroverkkojen ja koneoppimisen käyttö ei aina ole käytännöllistä. Vaikka pelihahmon voisi opettaa pelaamaan peliä optimaalisimmalla tavalla, se itsessään ei usein ole tavoite. Peleissä käytetään edelleen paljon tekniikoita, jotka ovat peräisin symboliselta aikakaudelta, kuten tilakoneita, polunetsintää ja asiantuntijajärjestelmiä. Syynä symbolisen aikakauden tekniikoiden hyödyntämiseen ovat pelien suhteellisen rajallinen monimutkaisuus verrattuna muihin sovelluksiin sekä symbolisten lähestymistapojen tarjoama selkeys ja ennustettavuus. Lisäksi symboliset järjestelmät verrattuna monimutkaisempiin koneoppimismenetelmiin ovat usein helpompia toteuttaa ja optimoida pelien tarpeisiin. Nämä tekniikat ovat kuitenkin kehittyneet pelien kehittyessä yhä paremmiksi vastaamaan pelien monimutkaisuutta ja muuttuneita tarpeita. (Millington, 2019, 7–10.) Pelivastustajilta halutaan usein realistista ja inhimillistä käyttäytymistä, ei pelkästään optimaalisia strategioita. Lisäksi on huomioitava prosessointiaika, sillä liiallinen monimutkaisuus voi hidastaa pelin sujuvuutta. Muutosten tekeminen tekoälyn toimintoihin on myös haasteellista, jos tekoäly on hyvin emergenttinen ja itseohjautuva. (Sizer, 2018.) On kuitenkin syytä huomioida, että kaikkia uusimpien tekoälytekniikoiden mahdollisuuksia ei ole vielä täysin hyödynnetty ja ala tarjoaa yhä tilaa uusille innovaatioille.

3.1 Historiaa

Tekoälyn tutkimus peleissä juontaa juurensa 1950-luvulle, jolloin ensimmäiset kokeilut tietokonepelien tekoälyn kanssa olivat vielä varsin yksinkertaisia. Aluksi tekoäly keskittyi erityisesti lautapeleihin, joissa sen pääasiallinen tehtävä oli pelata peliä mahdollisimman optimaalisesti ihmispelaajaa vastaan. Varhainen peleihin liittyvä tekoälytutkimus oli pääosin teoreettista, eivätkä siihen vaikuttaneet kaupalliset tavoitteet (History-Computer 2024). Varhaisissa peleissä tekoälyn rooli oli myös rajallinen, ja sen

toiminnot olivat enemmänkin toistuvia kaavoja kuin varsinaisesti älykästä käyttäytymistä (Schwab 2009, 3). Kuvio 2 esittää poimintoja pelien ja pelitekoälyn historian merkittävistä tapahtumista.



KUVIO 2. Aikajana pelien ja pelitekoälyn historiasta. (Schwab 2009, 4; Yannakakis & Togelius 2018, 8–21; Bedingfield 2023; Liat 2015; Mescarenhas 2017; Calvin 2020; Nnoli 2023)

1990-luvun lopulla ja 2000-luvun alussa arcade-pelikoneet, jotka olivat aiemmin olleet keskeinen osa pelimarkkinoita, alkoivat menettää markkinaosuuttaan kotitietokoneiden ja pelikonsolien kehittyessä ja yleistyessä. Koska kotikäyttöön tarkoitetut laitteet muuttuivat edullisemmiksi, helpommin saataviksi ja suorituskyvyltään entistä paremmiksi, myös pelinkehitys suuntautui yhä enemmän näille alustoille. Tällä oli merkitystä myös tekoälyn kehityksessä, sillä pelit monimutkaistuivat ja tekoälyyn liittyviä kokeiluja, ja algoritmien käyttöä alettiin soveltaa uusilla tavoilla ja entistä moninaisemmissa peligenreissä. (Yannakakis & Togelius 2018, 11.)

2000-luvulle tultaessa peliteollisuus siirtyi kohti dynaamisempaa tekoälyä, jossa päätöksenteko perustui emergentteihin, itsestään syntyviin ja ohjautuviin toimintamalleihin. Tämä tarkoitti, että tekoäly ei enää toistanut samoja kaavoja, vaan pystyi oppimaan ja sopeutumaan pelaajan toimintaan, luoden entistä haastavampia ja luonnollisempia pelikokemuksia. Tekoäly pystyi nyt käyttämään monimutkaisempia algoritmeja ja analysoimaan pelaajan valintoja antaen vastustajille entistä inhimillisempiä ja ennakoimattomampia piirteitä. Tekoälyä käytettiin esimerkiksi laajoja maailmoja sisältävissä avoimen

maailman peleissä, jotka toivat aivan uudenlaisen ulottuvuuden pelien tekoälyyn. Nämä pelit sisälsivät valtavia, eläväisiä pelimaailmoja, joissa tekoäly ei ainoastaan ohjannut vihollisten käyttäytymistä, vaan myös NPC-hahmojen päivittäistä elämää. Verkko- ja moninpelien suosion kasvaessa tekoälystä tuli myös entistä tärkeämpi pelaajien viihtyvyyden- ja tasapainon säilyttämisessä. (Schwab, 2009, 2; Northwood 2023.)

Viime vuosina tekoäly on kehittynyt huomattavasti erityisesti syväoppimisen ja neuroverkkojen myötä. Nämä teknologiat ovat mahdollistaneet esimerkiksi entistä älykkäämpien ja sopeutuvampien pelivastustajien luomisen. Syväoppimisen avulla pelit voivat myös generoitua dynaamisesti tarjoten uusia, aina muuttuvia pelimaailmoja ja kokemuksia. Tekoälyllä voidaan tehdä myös esimerkiksi entistä realistisempia grafiikoita. Tekoäly voi luoda henkilökohtaisempia pelikokemuksia, jotka mukautuvat pelaajan valintoihin ja pelityyliin. Lisäksi tekoäly voi auttaa pelaajaa itse pelissä, esimerkiksi tarjoamalla vinkkejä tai säätämällä pelin vaikeusastetta reaaliaikaisesti. Tulevaisuudessa tekoälyn ja pelien yhteinen kehitys saattaa avata vielä uusia peligenrejä ja innovatiivisia pelimuotoja, jotka tuovat entistä syvempää vuorovaikutusta ja immersiota peliin. Samalla nykyiset, perinteisemmät tekoälyn käyttötavat säilyvät, sillä kaikissa peleissä ei tarvitse aina käyttää kaikkein kehittyneimpiä teknologioita. (Schwab 2009, 2; Northwood 2023; Millington 2019, 6–9.)

3.2 Päätöksentekotekniikat

Päätöksentekotekniikat ovat menetelmiä, joita tekoäly käyttää valitakseen optimaalisimman toiminnan tietyssä tilanteessa. Vaikka tässä työssä päätöksentekoa tarkastellaan pääasiassa NPC-hahmojen näkökulmasta, päätöksentekotekniikoilla voidaan yhtä hyvin ohjata myös muita pelin osa-alueita, kuten ympäristön toimintaa, strategisia valintoja, tarinan kulkua ja vaikeustason säätämistä. Niitä voidaan soveltaa mihin tahansa tilanteisiin, joissa tekoälypohjaiset ratkaisut voivat olla tarpeellisia.

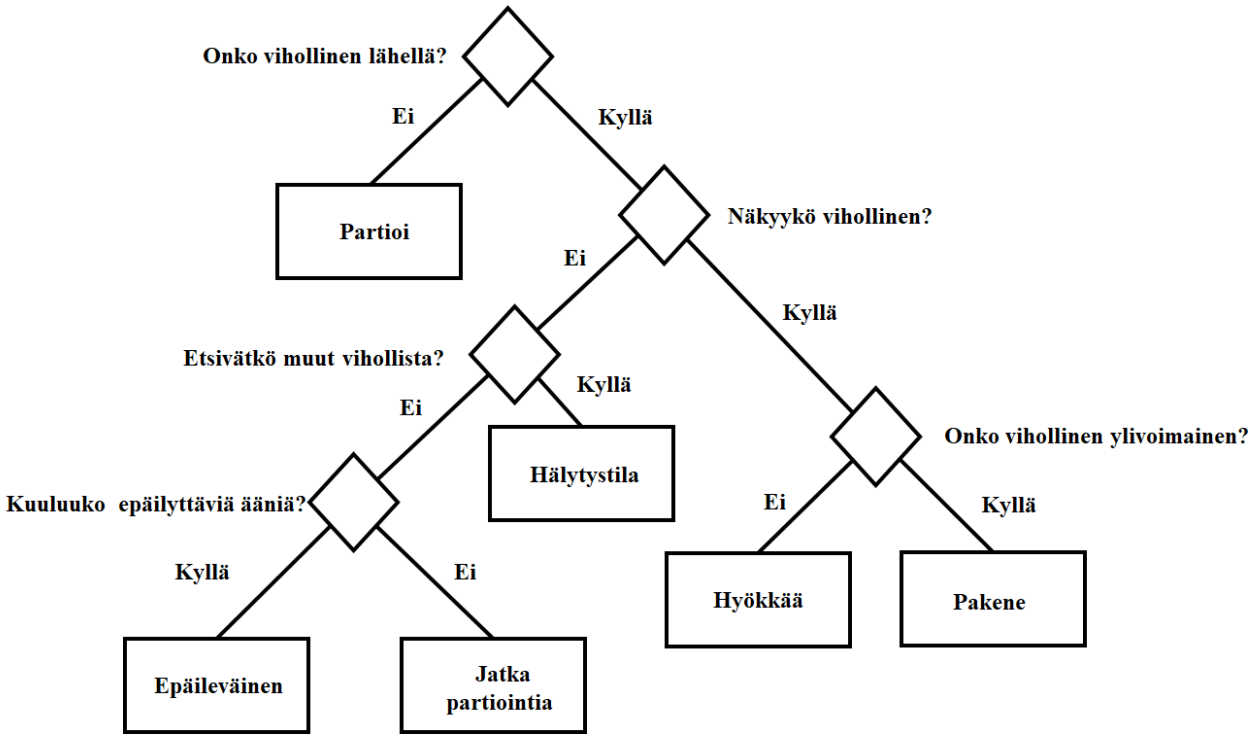
Kun suunnitellaan tekoälyä tietokonepeliin, ensimmäisiä vaiheita on määrittää, mitä kaikkea tekoälyn halutaan pelissä ohjaavan. NPC-hahmon tapauksessa tämä kattaa kaikki tarvittavat toiminnot, kuten liikkumisen, ampumisen tai tavaroiden keräämisen. Pelihahmon tulee pystyä joustavasti vaihtamaan eri toimintatilojen välillä tilanteen mukaan, jotta se voi reagoida pelin tapahtumiin ja ympäristöön halutulla tavalla. (Roberts 2023, 125.)

Erilaiset päätöksentekotekniikat ovat siis keskeisessä roolissa, kun peliin rakennetaan tekoälytoimintoja. Muita peleissä käytettyjä yleisiä tekniikoita ovat muun muassa reitinhakualgoritmit, kuten A*,

jotka auttavat tekoälyhahmoja navigoimaan pelimaailmassa. Erilaisissa peleissä tarvitaan erilaisten tekoälyalgoritmien soveltamista, joista tässä työssä esitellään yleisimpiä. Algoritmeja löytyy lisää esimerkiksi tämän työn lähdeteoksista, ja niitä voidaan soveltaa Godot-pelimoottorissa työssä esitetyillä tavoilla. Tekoälytekniikan valinnassa on otettava huomioon monia tekijöitä, kuten pelin genre, tekoälyn käyttötarkoitus, monimutkaisuus, laskentateho ja skaalautuvuus. Esimerkiksi monimutkaisessa strategiapelissä, jossa liikutellaan suuria määriä tekoälyhahmoja, tarvitaan todennäköisesti monimutkaisempia algoritmeja kuin yksinkertaisessa tasohyppelypelissä. (Millington 2000, 11–12.)

3.2.1 Tilakoneet

Tilakone (engl. *Finite-state machine*) on menetelmä, jossa määritellään kaikki mahdolliset tilat, joissa pelihahmo voi olla, sekä säännöt näiden tilojen välisille siirtymille. Kuviossa 3 on esitetty havainnollistava tilakone, joka voisi olla esimerkiksi osa tässä työssä toteutettavaa demopeliä. Tilakoneita käytettiin tekoälyn toteutuksessa erityisesti arcade-pelikoneiden aikakaudella ja 1990-luvun peleissä. (Roberts 2023, 125.) Yksi esimerkki varhaisesta tilakoneella toteutetusta tekoälystä on Pac-Man–pelissä, jossa viholliset jahtaavat pelaajaa. Vihollishahmoilla on useita eri tiloja, joita ne vaihtavat: jahtaus, pakeneminen ja kotiin palaaminen. Viholliskummitukset valitsevat tietyn kohdan, jota kohti ne aina kääntyvät kulmissa. Tilakone on suhteellisen yksinkertainen tekniikka toteuttaa, ja näin toteutettu pelin tekoäly saattaa vaikuttaa pelaajalle helposti ennakoitavalta, koska NPC-hahmo siirtyy aina samaan tilaan samoissa olosuhteissa. (Millington 2000, 8.)



KUVIO 3. Tilakaavio, joka kuvaa tilakoneen tilojen välisiä siirtymiä.

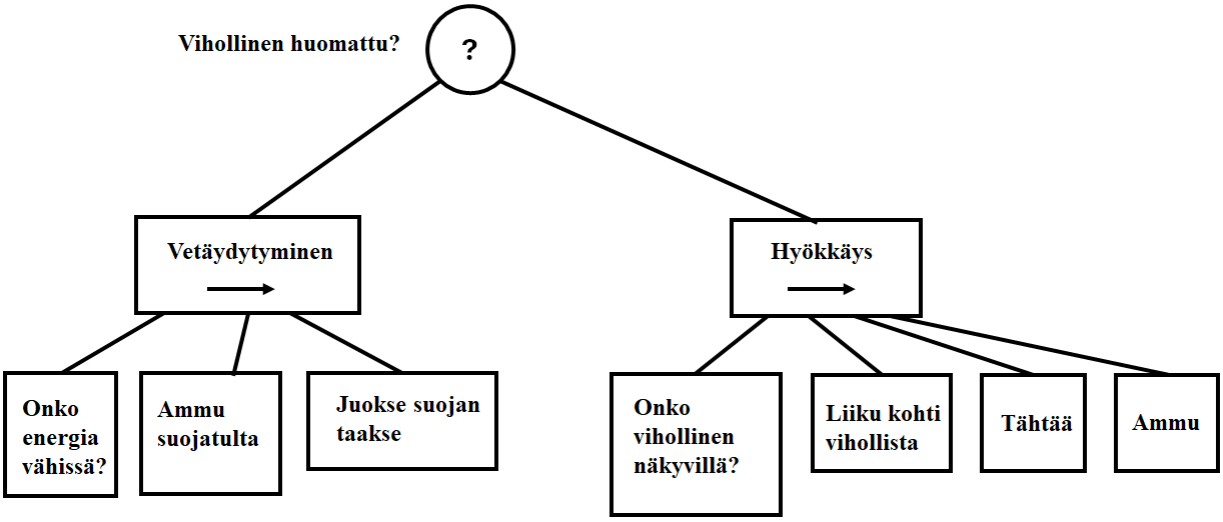
Tilakone-tekniikkaa voidaan laajentaa monin tavoin. Esimerkiksi tiloista voidaan tehdä hierarkkisia, jolloin hahmo voi saavuttaa samoja päämääriä useilla eri tavoilla. Tämä mahdollistaa monipuolisempia ja joustavampia päätöksentekotilanteita, joissa hahmo ei ole sidottu vain yhteen kiinteään toimintatapaan. Tilakone-tekniikan rajoituksiin, kuten yhdessä tilassa kerrallaan olemiseen ja muistin puutteeseen, on kehitetty erilaisia ratkaisuja. Esimerkiksi osittain tietyssä tilassa oleva tilakone (engl. *fuzzy state machine*) voi olla osittain useassa tilassa samaan aikaan. Muistin puutetta voidaan korjata luomalla yksinkertaisen muistipino, johon tallennetaan toiminnot, joita hahmo on aiemmin suorittanut. (Roberts 2023, 127.)

Tilakoneet ovat edelleen toimiva ja laajasti käytössä oleva tekniikka pelien kehityksessä. Tilakoneet voivat toimia hyvin yksinkertaisten, älykkäiltä vaikuttavien toimintojen kehittämisessä, mutta niistä tulee usein vaikeasti hallittavia suuremmissa projekteissa. Kuitenkaan kaikissa peleissä ei välttämättä tarvita monimutkaisempaa tekoälyä, vaan pelissä voi riittää yksinkertaisesti ympäristöön reagoiva vihollisvastustaja. Se voi olla joskus jopa toivottavaa, erityisesti silloin, kun osa pelin viehätystä on löytää ja hyödyntää vihollisen toistuvia käyttäytymismalleja. Joissain peleissä monimutkainen tekoäly olisi liikaa, ja vastustajan voidaan myös haluta olevan yksinkertainen ja helppo. Eli vaikka tilakoneet saattavat olla rajoittuneita tietyissä tilanteissa, ne ovat erityisen tehokkaita ja yksinkertaisia ratkaisuja

pelihahmojen käyttäytymisen hallintaan. Tilakoneita käytetään peleissä myös edelleen moniin tarkoituksiin, jotka ulottuvat tekoälyn toteuttamisesta paljon laajemmalle, kuten animaatioissa, ympäristön toiminnoissa, pelihahmojen tiloissa, käyttöliittymässä ja esimerkiksi pelin edistymisessä. Peleissä on usein monia toimintoja, jotka toimivat reagoivasti ja tilasta riippuvaisesti, ja tilakoneet tarjoavat tehokkaan tavan hallita näitä vuorovaikutuksia. Tilakoneita käytetäänkin usein yhdessä muiden tekoälytekniikoiden kanssa. (Roberts 2023, 125–128; Kirby 2011, 50–51.)

3.2.2 Käyttäytymispuut

Yleisesti käytetty ja merkittävä tapa kehittää tekoälyä peleissä on käyttäytymispuumenetelmä (engl. *Behaviour tree*), joka yleistyi 2000-luvun alussa ja vakiintui tärkeäksi osaksi peliteollisuuden tekoälyn kehitystä. Damian Isle laajensi hierarkkisen tilakoneen ideaa skaalautuvammaksi ja modulaarisemmaksi kehittäessään tekoälyä videopeliin *Halo 2*, jonka Bungie julkaisi vuonna 2004. Käyttäytymispuut mahdollistavat monimutkaisempien ja joustavampien päätöksentekoprosessien rakentamisen verrattuna perinteisiin lähestymistapoihin, kuten tilakoneisiin. (Roberts 2023, 130.) Käyttäytymispuut muistuttavat hierarkkisia tilakoneita, mutta ne yhdistävät useita tekoälytekniikoita, kuten aikataulutusta, suunnittelua ja toiminnan suorittamista. Käyttäytymispuun etuna on sen kyky käsitellä monimutkaisia tilanteita joustavasti ja selkeästi. Se perustuu hierarkkiseen rakenteeseen, jossa perusosina toimivat tehtävät, toisin kuin tilakoneiden tilat. Käyttäytymispuut ovat puumainen rakenne, jossa on eri lailla toimivia solmuja (KUVIO 4). Solmut voidaan jakaa kolmeen luokkaan: solmujen suoritusta hallitseviin, solmujen välistä tietoa prosessoiviin ja toimintoja edustaviin solmuihin. Yhteistä solmuille on, että ne palauttavat pääsolmulle tiedon tilastaan, joka voi olla suoritettu, käynnissä tai epäonnistunut. Tehtävät kootaan alipuiksi, jotka edustavat monimutkaisempia toimintoja. Koska tehtävillä on yhteinen rajapinta ja ne toimivat pääasiassa itsenäisesti, ne voidaan koota hierarkkisesti käyttäytymispuiksi, eikä tällöin tarvitse huolehtia siitä, miten puun alemmat tehtävät on toteutettu. (Millington 2019, 338.)



KUVIO 4. Käyttäytymispuun erilaisia solmuja.

Käyttäytymispuun suorittaminen on iteratiivinen ja jatkuvasti päivittyvä prosessi, joka tapahtuu tietyin aikavälein ja sopeutuu pelin tai ympäristön tilan muutoksiin (Roberts 2023, 130). Käyttäytymispuun rakenne vaihtelee tarpeiden mukaan, mutta yleisesti puun suoritus alkaa juurisolmusta, joka määrittää, mikä toiminta valitaan. Juurisolmu ohjaa suoritusta eteenpäin ja tarkastelee sen alaisia solmuja, jotka voivat olla muiden solmujen suoritusta ohjaavia, solmujen välistä tietoa prosessoivia tai toimintoja suorittavia. Muiden solmujen suoritusta ohjaavat solmut määrittävät suoritettavat toiminnat ja niiden järjestyksen. Solmujen välistä tietoa prosessoivat solmut voivat ennen toiminnan aloittamista tarkistaa olosuhteita, kuten ehtoja tai ajastuksia. Toimintosolmut, jotka sijaitsevat puun lehtisolmuissa, suorittavat varsinaiset toiminnot, kuten liikkumisen ja hyökkäämisen. Puun suoritus etenee alhaalta ylöspäin, ja jokainen solmu arvioi tilansa. Jos solmu suorittaa toiminnon onnistuneesti, se palauttaa onnistumisen tilan, ja vastaavasti jos toiminto epäonnistuu, palautetaan epäonnistumisen tila. Jos solmu on kesken, se pysyy ”käynnissä” eikä etene seuraavaan solmuun ennen kuin tilanne muuttuu. Tämä jatkuva arviointi ja päivitys mahdollistavat käyttäytymispuun sopeutumisen muuttuviin olosuhteisiin ja antavat pelissä toimivalle tekoälylle joustavan ja dynaamisen tavan reagoida ympäristön ja pelin tilanteisiin. (Millington 2019, 339.)

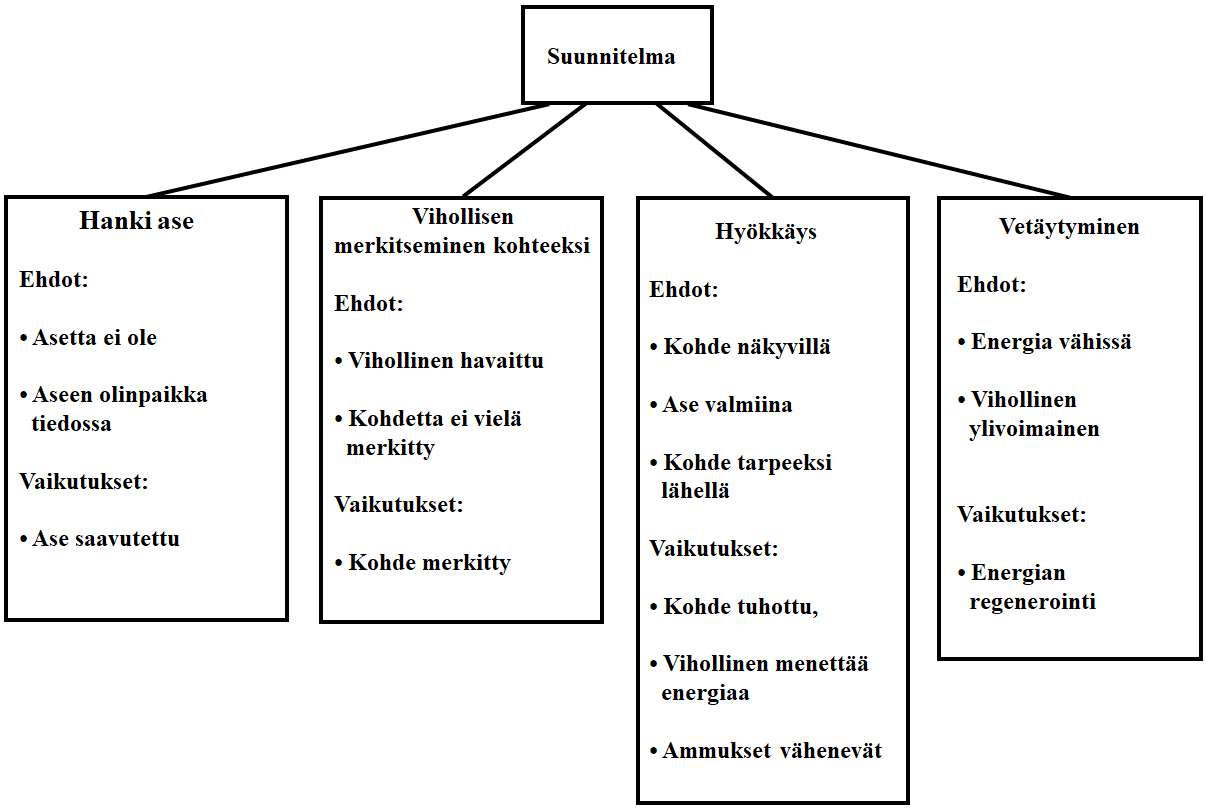
Käyttäytymispuiden rakentamisessa visuaalinen käyttöliittymä on usein selkeämpi ja tehokkaampi vaihtoehto kuin pelkkä koodipohjainen toteutus. Visuaalinen esitys tekee käyttäytymispuiden rakenteen intuitiivisesti ymmärrettäväksi, mikä helpottaa niiden suunnittelua. Muita käyttäytymispuiden etuja ovat laajennettavuus, uudelleenkäytettävyys sekä muokattavuus. Käyttäytymispuiden haasteena

on ollut niiden skaalautuvuus, ja erittäin laajoista puista voi tulla raskaasti suoritettavia. Tätä ongelmaa on kuitenkin voitu parantaa erilaisten solmurakenteiden avulla. Myös kaikkien mahdollisten tapahtumaskenaarioiden huomioiminen voi olla joskus työlästä. (Kirby 2011, 351; Millington 2019, 373; Endless Existence 2024.)

3.2.3 Tavoitepohjainen tekoäly

Tavoitteeseen suuntautunut käyttäytyminen (engl. *Goal-Oriented Behaviour*) on tekoälyn lähestymistapa, jossa tekoäly toimii tiettyjen tavoitteiden saavuttamiseksi. Tavoitepohjainen tekoäly poikkeaa perinteisistä reaktiivisista menetelmistä (kuten käyttäytymispuista) siinä, että tekoäly ei valitse toimintojaan pelkästään ympäristöstä saatujen syötteiden perusteella. Yksi esimerkki näin toteutetusta tekoälystä on GOAP (engl. *Goal-Oriented Action Planning*), joka tuli tunnetuksi erityisesti F.E.A.R.-pelistä vuodelta 2005, jossa sitä käytettiin onnistuneesti yhdessä tilakoneiden ja A*-hakualgoritmin kanssa. GOAP on ottanut vaikutteita STRIPS-suunnittelujärjestelmästä (*Stanford Research Institute Problem Solver*), joka kehitettiin 1970-luvulla. (Chaudhari 2017.)

GOAP:ssa tekoälyn ohjaamalle hahmolle annetaan tavoitteita, kuten ”tuhoa vihollinen” tai ”siirry toiseen huoneeseen”. Tekoäly pyrkii saavuttamaan tavoitteensa laatimalla suunnitelman, joka koostuu niitä tukevista toimista – kuvio 5 havainnollistaa tätä toimintamallia. Tavoitteet ovat priorisoituja, ja niiden tärkeysjärjestys muuttuu pelin kulun mukaan. Tekoäly luo toimintasuunnitelman ottaen huomioon pelin sen hetkisen tilanteen ja tavoitteensa. Tämän jälkeen se toteuttaa suunnitelman ja kun pelimaailma muuttuu, suunnitelma taas päivitetään. (Thompson 2020.)



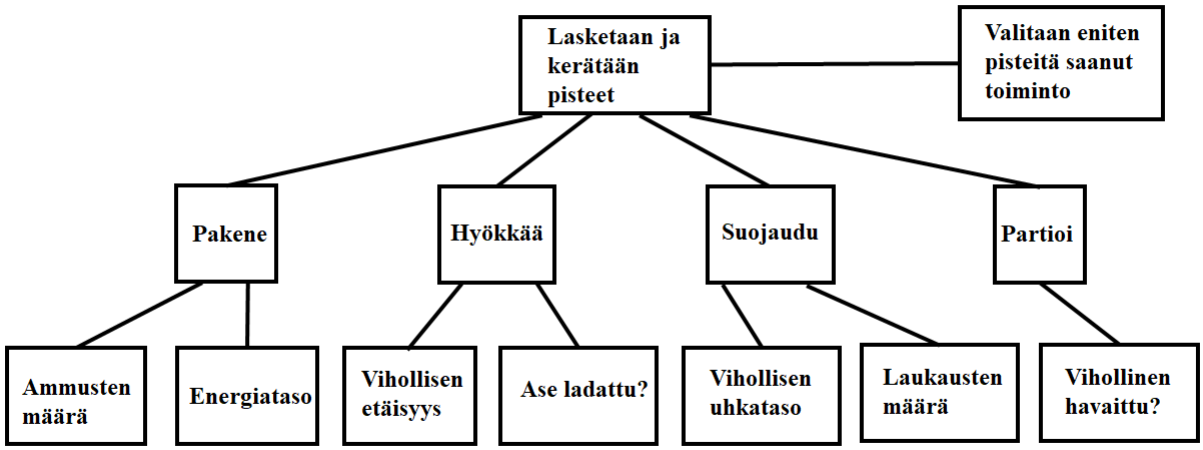
KUVIO 5. Esimerkki tavoitepohjaisen tekoälyn suunnitelman rakenteesta.

Myös tavoitepohjaisen tekoälyn käytössä tulee ottaa huomioon tiettyjä sille ominaisia puolia. Esimerkiksi suunnitelmien jatkuva uudelleen päivittäminen voi olla viedä laskentatehoja, etenkin jos hahmoja on paljon. Haasteita voi ilmetä myös virheiden jäljittämisessä ja korjaamisessa silloin, kun kaikki ei toimikaan odotetusti, sillä pelihahmo itse laatii suunnitelman suorittamistaan toiminnoista. GOAP toi pelien tekoälyyn dynaamisuutta ja älykkyyttä. Hyviä puolia tekniikassa on esimerkiksi se, että siihen voidaan helposti lisätä uusia tavoitteita ja toimintoja. (Toxigon Infinite 2025.)

3.2.4 Hyötypohjainen tekoäly

Utiliteettiteoria on ollut keskeinen käsite monilla tieteenaloilla jo pitkään, ja sitä on hyödynnetty muun muassa taloustieteessä (Read, 2004). 2000-luvun alussa suurimpaan hyötyarvoon pohjautuvasta tekoälystä (engl. *Utility AI*) on tullut yksi vaihtoehto perinteisille tekoälymenetelmille myös pelinkehityksessä. Tällainen tekoälytekniikka on nähtävissä esimerkiksi Maxisin kehittämässä *The Sims* -pelisarjassa, jossa hahmojen päätöksenteko pohjautuu tarpeiden ja tilanteiden arviointiin. *The Sims* -peleissä hahmoilla on tiettyjä tarpeita ja pelissä on erilaisia toimintoja, joista hahmo valitsee ne, jotka parhaiten täyttävät nämä tarpeet. (Brown 2023.)

Hyödyllisyyteen perustuvassa tekoälyssä lasketaan jokaiselle mahdolliselle toiminnalle numeerinen arvo, joka kuvaa sen hyödyllisyyttä. Hyötypisteet lasketaan ja johdetaan pelin muuttujista ja datasta jatkuvasti pelin kulun aikana. Tämän pistemäärän perusteella valitaan suoritettava toiminto, joka voi olla esimerkiksi korkeimmat pisteet kyseisellä hetkellä saanut toiminto. Hyötypohjaisen tekoälyn keskeinen toimintamalli on esitetty kuviossa 6. Kehittäjän tulee päättää, kuinka yksittäisen toiminnon hyötypisteet halutaan laskea ja mitä kaikkea dataa ja muuttujia hyötyarvon laskemisessa halutaan käyttää. Hyötypisteiden asettamisessa on tärkeää olla johdonmukainen, ja lisäksi niiden tulee olla samalla asteikolla, koska ne vertautuvat toisiinsa. Siksi normalisoidut pisteet, kuten arvot, jotka vaihtelevat 0–1 välillä, tarjoavat hyvän lähtökohdan ja ovat helposti verrattavissa. Esimerkiksi Graham kuvaa kirjassa *Game AI Pro*, kuinka shakissa voidaan arvioida kunkin siirron hyödyllisyyttä sen mukaan, kuinka vahvana pidämme siirtoa. Hyötypohjainen tekoäly pystyy tekemään ”parhaan arvauksen”, vaikka emme pystykään laskemaan kaikkia mahdollisia siirtoja. (Graham 2013, 113–114.)



KUVIO 6. Kuvio hyötypohjaisen tekoälyn päätöksentekoprosessista.

Hyötypohjaisen tekoälyn eduiksi voidaan nähdä helpompi suunniteltavuus: tekoälyn käyttäytymistä voi kuvata yksinkertaisilla säännöillä, kuten ”jos NPC on tulituksen alla, etsi suojaa”. Laajennettavuus on helpompaa, koska valmiiseen tekoälyyn voidaan lisätä sääntöjä ilman, että tarvitsee huolehtia monimutkaisista suhteista ja rakenteista. Yksinkertaisempi kehittämistapa saattaa myös parantaa tuottavuutta. Hyötypohjaisen tekoälyn haasteena voidaan mainita se, että älykkäiden päätösten tekeminen edellyttää pisteytyksen säätämistä, mikä voi viedä aikaa. Vaikka hyötypohjainen tekoäly pystyy toimi-

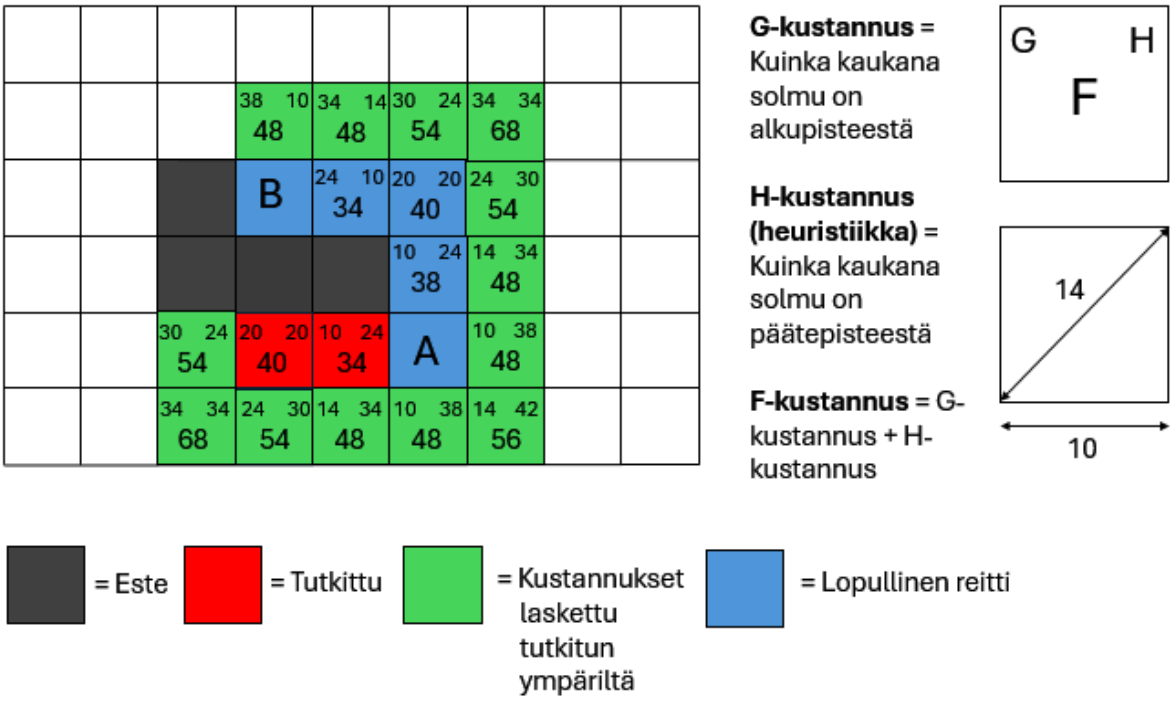
maan epätäydellisten tietojen perusteella ja tuottamaan älykkäältä vaikuttavaa tilanteisiin sopivaa käyttäytymistä, on tämä kuitenkin riippuvainen kehittäjän tekemistä valinnoista. Kun pelimaailmat ovat tulleet laajemmiksi, pelihahmojen määrä on kasvanut ja hahmojen käyttäytyminen monimutkaisemmaksi, voi hyötypohjainen tekoäly kuitenkin olla yksi vastaus näihin haasteisiin. (Rasmussen 2016.)

Hyötypohjaisessa tekoälyssä päätöksenteko perustuu erilaisten toimintavaihtoehtojen hyötyarvojen laskentaan. Tekoäly valitsee toiminnon, joka maksimoi kokonaisarvon annetussa tilanteessa. Tämä lähestymistapa mahdollistaa joustavan ja tilannetietoisen toiminnan, mutta voi myös tehdä tekoälyn käyttäytymisestä osittain ennakoimatonta erityisesti monimutkaisissa tai nopeasti muuttuvissa ympäristöissä. Tämän vuoksi hyötyfunktioiden huolellinen suunnittelu ja järjestelmän testaaminen ovat keskeisiä osia kehitysprosessia. Hyötypohjaisen tekoälyn merkittäviä etuja ovat muun muassa sen kyky reagoida nopeasti ympäristön muutoksiin ja tuottaa käyttäytymistä, joka vaikuttaa älykkäältä ja vaihtelevalta. Tekoälyn toiminnot eivät ole tiukasti sidottuja toisiinsa, mikä parantaa järjestelmän modulaarisuutta ja helpottaa uusien toimintojen lisäämistä tai poistamista ilman suuria muutoksia järjestelmärakenteeseen. Lisäksi hyötyarvojen laskenta voidaan toteuttaa laskennallisesti kevyesti, esimerkiksi käyttämällä liukulukuarvoja (engl. *float*), mikä vähentää järjestelmän kuormitusta ja mahdollistaa skaalautuvan toteutuksen. Koska hyötylaskenta ei ole sidottu yksittäisiin koodikomponentteihin, se mahdollistaa tekoälyn säätämisen myös järjestelmissä, joissa kehittäjällä ei ole täyttä hallintaa kaikkeen lähdekoodiin. (Toxigon Infinite 2025; AI and Games 2025.)

Hyötypohjainen tekoäly voi olla erityisen hyödyllinen sellaisessa pelikehityksessä, jossa ohjataan useita samanaikaisia toimintoja – kuten NPC:n päätöksentekoa, strategista käyttäytymistä tai pelimaailman dynaamista sisällönhallintaa. Usein käytössä on useita tekoälykomponentteja, jotka toimivat yhdessä mutta itsenäisesti, mahdollistaen kompleksisen ja luonnolliselta vaikuttavan pelimaailman. On kuitenkin huomioitava, että vaikka hyötypohjainen algoritmi tarjoaa lukuisia etuja, sen toiminnan laatu ja tehokkuus riippuu vahvasti kehittäjän tekemistä valinnoista. Tekoälyn suunnittelussa tulee miettiä, kuinka syvällisesti se ohjaa eri toimintoja, kuinka paljon päätöksentekoa annetaan tekoälylle ja kuinka pelimaailman data integroidaan hyötylaskentaan. Hyötypohjainen tekoäly on kuitenkin hyvä vaihtoehto käyttäytymispuille, jotka ovat pelialalla suosittu tekoälyn kehitystapa. (Toxigon Infinite 2025; AI and Games 2025.)

3.3 A*-polunetsintä algoritmi

A*-algoritmi (engl. *A-star*) on tehokas ja laajasti käytetty polunetsintäalgoritmi, jota hyödynnetään esimerkiksi peleissä, robotiikassa ja karttasovelluksissa. Algoritmin perusidea kehitettiin jo 1960-luvulla, mutta siitä on tullut eräänlainen alan standardi erityisesti 1990-luvulta alkaen, jolloin sitä alettiin käyttää laajemmin peleissä. A*:n tavoitteena on löytää lyhyin tai edullisin reitti lähtöpisteestä kohteeseen, ottaen huomioon esteet ja mahdolliset kulkukustannukset. Peleissä A*-algoritmia käytetään esimerkiksi hahmojen liikkumiseen kartalla esteiden ohi esimerkiksi silloin, kun vihollinen etsii reittiä pelaajan luo. Algoritmi on myös hyödyllinen esimerkiksi monimutkaisissa kentissä liikkumisessa sekä strategiapeleissä yksiköiden liikuttamisessa.



KUVA 1. Esimerkki A*-algoritmin toiminnasta (mukaillen Lague 2014).

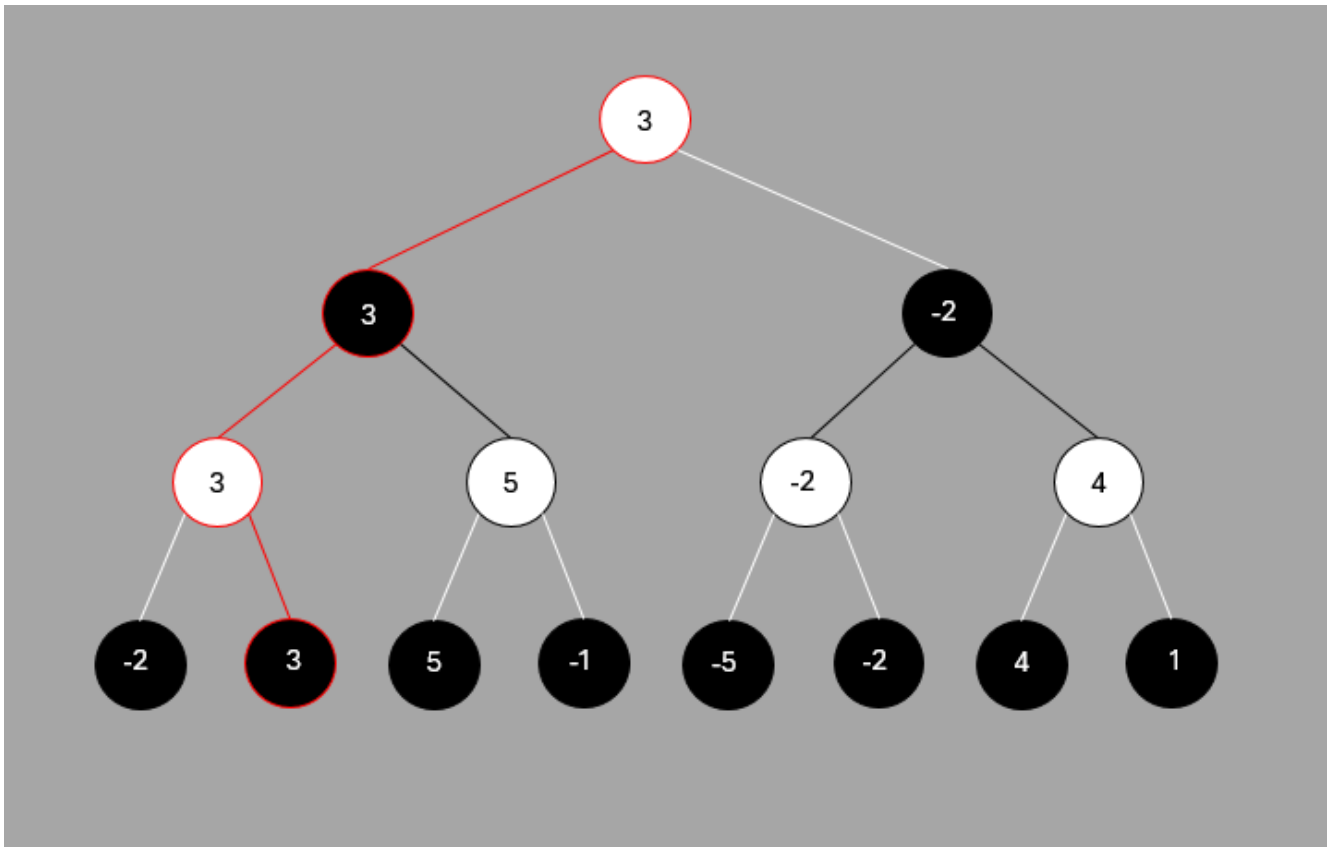
Algoritmi etsii lyhyimmän reitin ottamalla huomioon sekä kuljetun matkan että arvioidun etäisyyden kohteeseen. Se valitsee aina seuraavaksi tutkittavaksi sen solmun, joka vaikuttaa parhaalta vaihtoehdolta – eli joka on sekä lähempänä maalia että kulkee reittiä, jonka kokonaiskustannus vaikuttaa pieneltä. Tätä prosessia toistetaan, kunnes maali saavutetaan ja reitti palautetaan. Kuvassa 1 havainnollistetaan tätä prosessia: algoritmi on etsinyt lyhyimmän reitin pisteestä A pisteeseen B ottaen huomioon

kustannukset. A*-algoritmin tehokkuus perustuu heuristiikan käyttöön, mikä ohjaa etsintää kohti maalia. Heuristiikka vaikuttaa merkittävästi suorituskykyyn – mitä parempi arvio tulevasta reitistä, sitä nopeammin ja tehokkaammin algoritmi toimii. A*:n etuja ovat sen helppo toteutettavuus, tehokkuus ja optimointimahdollisuudet. (Millington 2019, 215–227.)

3.4 Minimax-algoritmi

Minimax on klassinen hakualgoritmi, jota voidaan käyttää esimerkiksi strategiapeleissä (shakki, ristinolla). Sen tavoitteena on löytää paras mahdollinen siirto sillä oletuksella, että vastustaja pelaa myös optimaalisesti. Minimax-algoritmi pyrkii valitsemaan parhaan mahdollisen siirron peleissä, jossa molemmilla on sama tavoite, ja lisäksi peli on niin sanottu ”täydellisen informaation peli”, eli jokaisen siirron mahdollinen vaikutus lopputulokseen on mahdollista tietää. Sen yhtenä merkittävänä kehittäjänä pidetään Claude Shannonia 1950-luvulla. Samankaltaisia muita minimaxin jälkeen kehittyneitä algoritmeja ovat esim. *negamax*, *expectimax* ja *monte carlo* -puuhaku. (Roberts 2023, 195.)

Minimax-algoritmissa siirrot esitetään puumaisena rakenteena, jossa kuvataan kaikki mahdolliset siirrot tiettyyn syvyyteen asti. Tavoitteena on minimoida vastustajan etua ja vastaavasti taas maksimoida omaa etua. Puussa solmut edustavat yhtä mahdollista siirtoa, ja ne haarautuvat yhä suurempiin määriin mahdollisia siirtoja (KUVIO 7). Kuviossa 7 musta pyrkii minimoimaan arvoa eli kohti negatiivista, kun taas valkoinen tavoittelee maksimaalista arvoa eli positiivista suuntaa. Algoritmi käy puun läpi ja valitsee siirron, joka maksimoi pelaajan edun olettaen, että vastustaja pyrkii tekemään myös parhaat mahdolliset siirrot. Minimax on suhteellisen yksinkertainen algoritmi, joka voi muuttua raskaaksi suorittaa puun kasvaessa liian laajaksi. Sillä on monia rajoituksia, joita on pyritty myös paikkaamaan uudemmissa algoritmeissa. Rajoituksia ovat muun muassa pelitilojen suuri määrä, soveltumattomuus epätäydellisen tiedon peleihin ja oppimattomuus. (Millington 2019, 747–750, 753.)



KUVIO 7. Esimerkki minimax-algoritmista, jossa jokainen solmu kuvaa yhtä siirtoa ja sen arvoa.

3.5 Neuroverkot

Viime vuosina pelialalla on alettu ottaa käyttöön myös uudempia teknologioita, kuten keinotekoisia neuroverkkoja ja vahvistusoppimista. Niitä on hyödynnetty menestyksekkäästi monilla tavoin: esimerkiksi prosessien optimoinnissa, liikenteessä ja kielen- ja kuvantunnistuksessa. Tekoälyneuroverkot eivät ole yksittäisiä algoritmeja, vaan pikemminkin yhdessä toimivia, kerroksittaisia rakenteita, jotka pystyvät käsittelemään monimutkaisia tietosyötteitä ja oppimaan niiden perusteella. Neuroverkot toimivat usein muiden algoritmien rinnalla tai osana laajempaa järjestelmää, ja niiden avulla voidaan mallintaa sellaisia ongelmia, joita perinteisillä menetelmillä on vaikea ratkaista. Neuroverkkoja on olemassa monenlaisia, ja niiden rakenne ja toimintaperiaatteet vaihtelevat käyttötarkoituksen mukaan. (Millington 2019, 642–646; Hendricks 2019.)

Neuroverkkoja hyödynnetään peleissä erityisesti sellaisissa tilanteissa, joissa halutaan saavuttaa luonnollinen, sopeutuva ja oppiva käyttäytyminen. Yksi keskeinen käyttökohde on pelihahmojen toiminnan ohjaaminen. Neuroverkkojen avulla NPC-hahmot voivat oppia reagoimaan pelaajan toimintaan, teke-

mään taktisia päätöksiä ja käyttäytymään ennakoimattomammin verrattuna perinteisiin, ennalta määriteltyihin sääntöihin perustuviin algoritmeihin. Toinen tärkeä sovellusalue on pelaajan käyttäytymisen analysointi. Neuroverkot voivat oppia tunnistamaan pelaajan pelityylin ja mukauttamaan pelin vaikeustasoa tai tapahtumia sen mukaan. Näin pelaamisesta saadaan yksilöllisempi ja immersiivisempi kokemus. Lisäksi neuroverkkoja käytetään muun muassa dynaamiseen sisällöntuotantoon, grafiikan parantamiseen sekä pelitasojen, ympäristöjen ja tehtävien automaattiseen luomiseen. Neuroverkot voivat mahdollistaa pelien sisällön jatkuvan vaihtelun ja uusien kokemusten tarjoamisen ilman manuaalista suunnittelua. Neuroverkkoja hyödynnetään myös kilpailukykyisten tekoälyvastustajien kehittämisessä. Esimerkiksi vahvistusoppimisen avulla tekoäly voi opetella pelaamaan peliä itseään vastaan miljoonia kertoja, kunnes se saavuttaa erittäin korkean tason suorituskyvyn. Tämän kaltaista lähestymistapaa on käytetty muun muassa strategia- ja taistelupeleissä sekä e-urheilun harjoitusvastustajissa. (Fink 2023; Hendricks 2019.)

Neuroverkoilla on kuitenkin omat heikkoutensa, ja niiden käyttöönotossa pelien tekoälyssä on omat hankaluutensa. Neuroverkot kulkevat vastaan sääntöä, että tekoälyn tulisi olla täysin kehittäjän hallittavissa, jolloin lopputuloksesta voi tulla arvaamaton (Roberts 2023, 280). Toinen ongelma on, että tekoälystä saattaa tulla laskennallisesti raskas. Tekniikan toteuttaminen voi olla myös epäkäytännöllistä ja liian monimutkaista yksinkertaisiin tehtäviin. Monimutkaisten tilanteiden hallitseminen voi olla vaikeaa. Erilaiset neuroverkkoihin perustuvat tekniikat ovat kuitenkin löytäneet käyttötarkoituksia, ja on todennäköistä, että teknologiat kehittyvät edelleen tarjoten paremman soveltuvuuden peleihin. (Hendricks 2019.)

4 GODOT

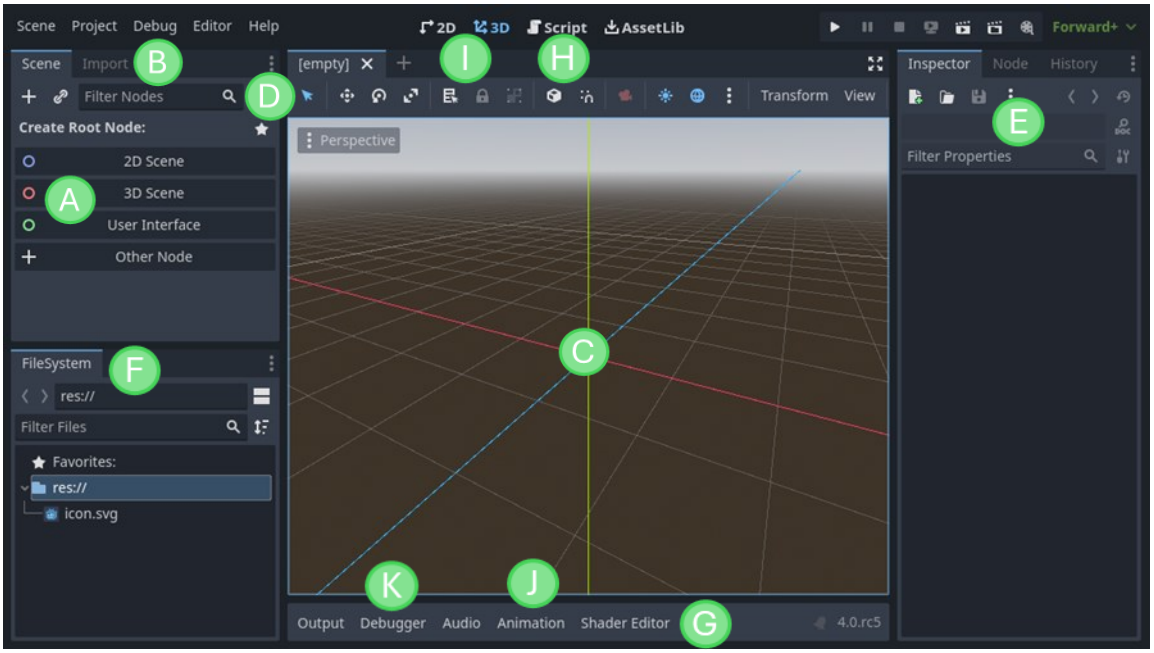
Godot on ilmainen ja avoimen lähdekoodin pelimoottori, joka julkaistiin MIT-lisenssillä vuonna 2014 (Godot documentation, n.d.). Godotin kehitys alkoi sisäisenä projektina vuonna 2007, kun Juan Linietsky ja Ariel Manzur loivat sen itselleen tilauksesta tehtyjen pelien tuottamiseen. Nykyisin Godotin kehitys on aktiivisen yhteisön ohjaamaa ja vapaaehtoisten kehittäjien toteuttamaa. Godotin kehitystä tukee voittoa tavoittelematon Godot Foundation -säätiö. Todennäköisesti erityisesti Godotin lisensoinnin ja maksuttomuuden vuoksi se on saanut suosiota etenkin indie-pelinkehittäjien keskuudessa. (Godot Engine, n.d.) Esimerkiksi *Global Game Jam 2024* -tapahtumassa Godot oli noussut toiseksi suosituimmaksi pelimoottoriksi Unityn jälkeen (GameFromScratch, 2024).

Godot-pelimoottoria voidaan käyttää useilla eri alustoilla: Windowsilla, macOS:llä, Linuxilla, Androidilla, iOS:llä sekä web-selaimilla. Kehittämistä tuetaan monille alustoille, kuten Windows, macOS, Linux, Android, iOS ja HTML5. Lisäksi Godot tukee pelien vientiä konsolialustoille, mutta näiden käyttö edellyttää lisenssejä ja erityistä kehityspääsyä. Yhtenä syynä sille voidaan pitää sitä, että Godot on avoimen lähdekoodin moottori, ja konsolien ohjelmistokehityspaketit ovat salaisia eli niitä ei voitaisi julkaista avoimella lähdekoodilla. (Godot documentation, n.d.)

Godot tukee useita ohjelmointikieliä koodiskriptien kirjoittamisessa. Pääasiallisena kielenä käytetään Godotin omaa *GDScriptiä*, joka muistuttaa Pythonia ja on helppokäyttöinen erityisesti aloittelijoille. Toinen vaihtoehto on C#. Lisäksi Godotissa voi käyttää C++:aa, mutta tämä vaatii erillisen työkalun käyttöä. (Godot Documentation, n.d.) Versiosta 3.0 eteenpäin (vuodesta 2017 alkaen) Godotissa kokeiltiin myös omaa *VisualScript*-ohjelmointikieltä, mutta se poistettiin Godot 4.0 -versiossa vuonna 2022. VisualScript saattaa kuitenkin jatkaa kehitystään omana moduulinaan tulevaisuudessa. (Linietsky, 2022.)

4.1 Godotin käyttöliittymä

Godot Engine on monipuolinen pelimoottori, joka tukee sekä 2D- että 3D-pelien luomista yhtenäisestä käyttöliittymästään käsin. Moottori tarjoaa laajan työkalupaketin, joka kattaa kaikki pelinkehityksen perustoiminnot. Lisäksi Godot mahdollistaa käyttäjien luomien lisäosien lataamisen ja tarvittaessa myös omien lisäosien kehittämisen, joka tekee Godotista joustavan käyttää. Kuvassa 2 on esitetty Godotin käyttöliittymän keskeisiä osia ja sen toiminnallisuuksia. (Godot documentation, n.d.)



KUVA 2. Godotin käyttöliittymä (Godot documentation n.d.).

A. Skenepuu (engl. *Scene Tree*) -näkymä: Näyttää nykyisen skenen kaikkien objektien hierarkian. Se sijaitsee vasemmalla puolella käyttöliittymää ja sillä voi lisätä, poistaa ja muokata nodeja.
B. Työkalupalkit: Nämä löytyvät käyttöliittymän yläosasta ja sisältävät tärkeitä työkaluja skenen muokkaamiseen (uusien objektien luominen, projektin tallentaminen, kumoamis- ja uudelleentekemistoiminnot).
C. Skenenäkymä (engl. *Scene View*): Keskellä oleva alue, jossa muokataan skenejä visuaalisesti. Näkymässä voi siirtää, kiertää ja skaalata objekteja, katsella ja navigoida pelimaailmassa sekä 2D- että 3D-tilassa.
D. Ylätason työkalut (engl. *Overlay Tools*): Skenenäkymässä olevat työkalut auttavat yksittäisten objektien muokkaamisessa, kuten siirrossa, pyörityksessä ja skaalaamisessa. Nämä työkalut auttavat tekemään interaktiivisia muutoksia suoraan skeneihin.
E. *Inspector*-ikkuna: Oikealla puolella sijaitseva ikkuna, jossa näkyvät valitun objektin ominaisuudet. Täällä voi muokata muuttujia ja komponentteja (sprite-nodejen tekstuureja, ominaisuuksia tai skriptejä) suoraan.
F. Projekti-ikkuna: Tässä hallitaan projektin tiedostoja, kuten skriptejä, skenejä, resursseja ja muita tiedostoja. Tässä ikkunassa voi järjestää ja käyttää kaikkia projektin sisältöjä.
G. Tilapalkki (engl. *Status Bar*): Käyttöliittymän alareunassa oleva palkki, joka näyttää tietoa, kuten nykyisen kuvanopeuden (FPS), konsolin tulosteen sekä aktiiviset debuggaus- tai animaatiotyökalut.

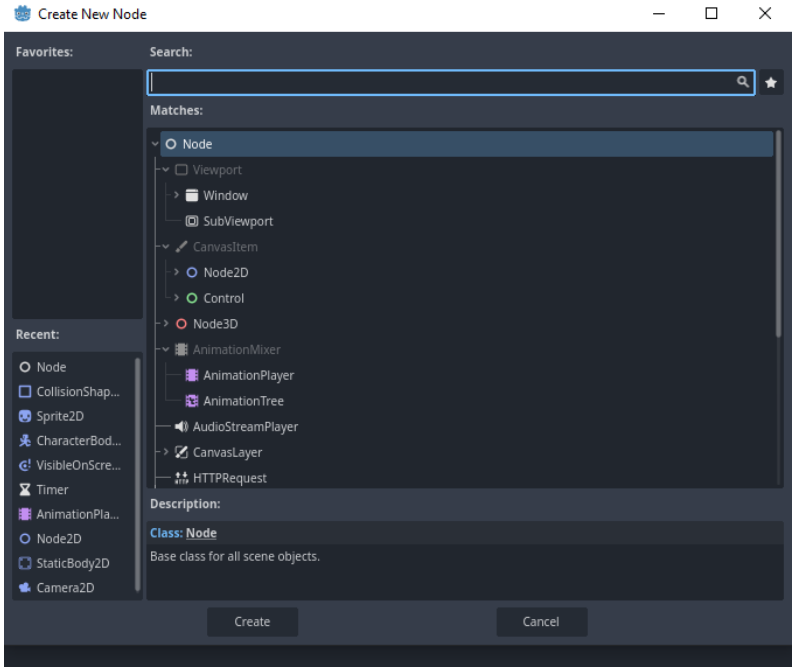
H. Koodieditori: Sisäänrakennettu editori GDScriptin tai muiden tuettujen kielten, kuten C#:n ja VisualScriptin, kirjoittamiseen. Sisältää ominaisuuksia kuten syntaksin korostus, automaattinen täydennys ja virheenkorjaustyökalut.
I. Tasoeditorit: Godotissa on sekä 2D- että 3D-tasoeditorit, jotka auttavat pelimaailman luomisessa. Työkalut on räätälöity kummankin tasoeditorin tarpeisiin.
J. Animaatiotyökalut: Nämä työkalut auttavat animoimaan objekteja, hahmoja ja skenejä avainruutujen, käyrien ja aikaratojen avulla. Voit helposti luoda ja hallita animaatioita sekä 2D- että 3D-objekteille.
K. Virheenkorjaustyökalut: Sisältää virheenkorjausominaisuuksia, kuten ajonpysähtymispisteet (engl. *Breakpoints*), pinojen jäljityksen (engl. *Stack Trace*) ja suorituskyvyn seurannan, joiden avulla voidaan tunnistaa ja korjata koodin ongelmia. (Godot documentation n.d.)

4.2 Godotin suunnittelufilosofiaa

Kaikki pelimoottorit perustuvat abstraktioihin, jotka helpottavat pelinkehitystä ja piilottavat monimutkaisempien teknisten yksityiskohtien käsittelyn kehittäjältä. Godotissa nämä abstraktiot rakentuvat muutamien keskeisten konseptien ympärille, joita ovat *node, scene, scene tree* ja *signal*. Ne määrittävät pelin rakennetta ja kehittämisen tapaa. Godot seuraa oliopohjaisen ohjelmoinnin (OOP) paradigmaa, eli että kaikki pelin objektit ja toiminnot voidaan käsitellä oliomuotoisina yksiköinä, jotka voivat periytyä ja vuorovaikuttaa toistensa kanssa. Tämä tekee pelinkehityksestä intuitiivisempaa ja helpottaa monimutkaisempien pelirakenteiden luomista. (Godot documentation, n.d.)

Godotin keskeisimpiä konsepteja on noodi (engl. *node*), joka toimii peruskomponenttina pelin elementtien hallinnassa ja toimintojen rakentamisessa. Noodit ovat yksinkertaisia objekteja, mutta niiden yhdistämisen ja periytymisen avulla voidaan luoda monimutkaisempia ja dynaamisempia rakenteita. Noodit muistuttavat ohjelmointikielissä käytettyjä luokkia ja objekteja. Noodi-järjestelmässä kukin noodi edustaa pelin elementtiä, kuten hahmoja, esineitä tai ympäristön osia, ja niillä on omat ominaisuutensa ja käyttäytymisensä. Kuten luokat ja objektit ohjelmointikielissä, noodi voi olla erikoistunut tehtäväänsä ja periä toiminnallisuuksia muilta noodeilta, mikä mahdollistaa jäsennellyn ja modulaarisen pelinrakenteen. Tämä järjestelmä auttaa kehittäjiä organisoimaan pelin eri osia ja hallitsemaan niiden välistä vuorovaikutusta selkeästi ja joustavasti. Mikäli valmiit toiminnot eivät riitä, kehittäjä voi laajentaa tai luoda uusia noodeja, jotka palvelevat tiettyjä pelin tarpeita (KUVA 3). Tällainen joustava

kehitystapa antaa kehittäjille vapauden mukauttaa järjestelmää ja jatkaa sen rakentamista omien visioidensa mukaan. (Godot documentation, n.d.) Tällaista toteutustapaa hyödynnetään myös tässä opinnäytetyössä, kun demosovelluksen tekoälyn toteutuksesta on pyritty tekemään siitä modulaarinen, helposti muutettava ja jatkettava.

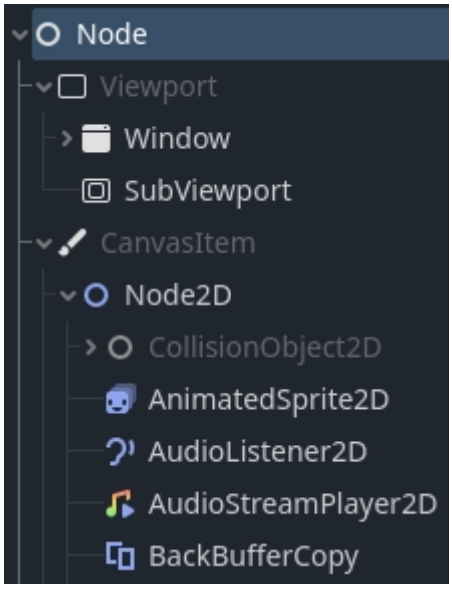


KUVA 3. Kuvakaappaus uuden noodin luomisesta Godot-editorissa. (Godot documentation, n.d).

Skene puolestaan on kokoelma aiemmin mainittuja noodeja. Se toimii ikään kuin pelin ”säiliöinä” ja voi sisältää pelin kohtauksia, hahmoja, valikoita tai mitä tahansa pelissä tarvittavaa sisältöä. Skenet tallennetaan Godotissa *tscn*-tiedostoina, ja ne ovat joustavasti pelinkehittäjän hyödynnettävissä. Esimerkiksi yksi skene voi olla kokonainen pelikohtaus, mutta se voi myös olla yksittäinen hahmo tai pelin valikko. Godotissa on myös helppo käyttää skeneä pelin rakenteen jakamiseen, ja ne voivat toimia uudelleen käytettävinä rakennuspalikoina. Tämä tekee pelin kehittämisestä joustavaa ja tehokasta, sillä kehittäjä voi järjestää ja hallita pelin sisältöä haluamallaan tavalla. (Godot documentation n.d.)

Skenepuu organisoi kaikki pelin objektit hierarkkisesti. Tämä puumainen rakenne auttaa pelinkehittäjää hallitsemaan pelin eri osia tehokkaasti ja luo selkeän tavan järjestää objekti- ja komponenttirakenteet. Skenepuu muodostuu erilaisista noodeista rakentuvista komponenteista, jotka voivat olla toistensa lapsia tai vanhempia (KUVA 4). Objektit voivat periä ominaisuuksiaan ja toimintojaan noodeilta, joiden lapsia ne ovat. Tämä hierarkkinen rakenne tekee pelin objektien hallinnasta intuitiivisempaa, koska sen avulla kehittäjä voi helposti nähdä, kuinka pelin elementit liittyvät toisiinsa ja kuinka niiden

tilaa voidaan hallita. Skenepuu on olennainen osa pelin logiikan ja muiden elementtien yhdistämistä. (Godot documentation, n.d.)



KUVA 4. Kuvakaappaus skenepuusta (Godot documentation, n.d).

Godot-pelimoottori tarjoaa sisäänrakennettuja valmiita rakenteita, kuten luokkia ja noodeja, jotka ovat keskeisiä uusien toimintojen kehittämisessä. Godotin pohjaluokkia voidaan laajentaa ja muokata skriptien avulla, jolloin luodaan mukautettuja toimintoja pelin tarpeisiin. *Object*-luokka on perusluokka, josta melkein kaikki muut luokat, kuten noodiluokkakin, periytyvät. *Skripti* taas on Object-perusluokan ominaisuus, joka mahdollistaa luokkien toiminnan määrittelyn, laajentamisen ja mukauttamisen. Käytännössä tämä näkyy niin, että Godotissa objekteihin voidaan liittää skriptejä, jotka määrittelevät sen käyttäytymistä, ja joilla sen toiminnallisuuksia voidaan laajentaa ja muokata. Skripteissä käytetään *extends*-avainsanaa, kun halutaan määrittää, että skripti laajentaa tai perii tietyn noodin toiminnallisuuden. Hyötypohjaisen tekoälyn kehittäminen tässä työssä perustuu näihin perusperiaatteisiin. Monet hyvät ja toimivat käytännöt Godotissa pohjautuvat oliopohjaisen suunnittelun hyödyntämiseen yhdessä skenejen, noodien ja skriptien kanssa. (Godot documentation, n.d.)

Signaalit, jotka mahdollistavat eri pelin osien kommunikoinnin keskenään, ovat myös keskeinen konsepti Godotissa. Signaalit toimivat kuin viestinviejät, jotka ilmoittavat tapahtumista pelissä. Kun jokin tapahtuma – kuten käyttäjän syöte tai pelin logiikka – laukaisee signaalin, toiset objektit, jotka ovat kuuntelemassa tätä signaalia, voivat reagoida siihen suorittamalla halutun toiminnon. Signaalit ja niiden kuuntelu muodostavat dynaamisen järjestelmän, mikä mahdollistaa pelin reagoinnin erilaisiin ta-

pahtumiin. Signaalit ovat yksi tapa yhdistää objekteja ja skriptejä toisiinsa. Objektit voivat kutsua toistensa toimintoja suoraan, tai ne voivat olla osa ryhmiä, joille voidaan lähettää komentoja yhdellä kertaa, tai *autoload*-ominaisuuden avulla objekteista voi tehdä globaaleja. (Godot documentation n.d.)

4.3 Tekoälytoimintojen käyttö Godotissa

Godotin perusversiossa ei ole valmiina laajasti työkaluja varsinaisten tekoälyalgoritmien kehittämiseen. Toisaalta Godotin valmiit toiminnot ja mahdollisuus laajentaa moottoria lisäosien avulla tarjoavat joustavan ja muokattavan pohjan erilaisten tekoälytoimintojen rakentamiseen. Moottorin sisäänrakennettu *NavigationServer* tarjoaa *NavMesh*-pohjaisen ratkaisun ei-pelattavien hahmojen liikuttamiseen sekä 2D- että 3D-peleissä. Navigointiin liittyviä toimintoja varten Godot sisältää valmiita nooodeja, kuten *NavigationMeshInstance*, *Navigation* ja *PathFollow*, joiden avulla voidaan toteuttaa esimerkiksi A*-algoritmiin perustuva reitinhaku. (Godot Documentation, n.d.).

Tekoälyn toteuttaminen Godot-pelimoottorissa on joustavaa ja monipuolista. Kehittäjällä on mahdollisuus valita, luodaanko tekoälyratkaisut itse esimerkiksi GDScript-ohjelmointikielellä vai hyödynnetäänkö valmiita ratkaisuja, kuten plugineita tai kolmannen osapuolen kirjastoja. Nämä ulkoiset työkalut voivat sisältää valmiiksi toteutettuja tekoälymalleja ja algoritmeja, jotka nopeuttavat kehitysprosessia ja vähentävät manuaalisen työn tarvetta. Valmiit pluginit voivat sisältää muun muassa käyttäytymispuita (*behavior trees*), tilakoneita (*state machines*) tai muita tekoälytekniikoita, jotka voidaan helposti integroida osaksi projektia ilman, että järjestelmää täytyy rakentaa kokonaan alusta. Moniin näistä ratkaisuista on mahdollista liittää käyttöliittymiä, jotka helpottavat tekoälyn säätämistä ja testaamista kehitystyön aikana. Tämä joustavuus antaa kehittäjille mahdollisuuden joko rakentaa täysin räätälöity tekoälyjärjestelmä tai hyödyntää olemassa olevia yhteisön tuottamia työkaluja, mikä nopeuttaa kehitystyötä huomattavasti. Godotiin on mahdollista integroida ulkoisia tekoälykirjastoja, kuten *TensorFlow* tai *PyTorch*, jolloin voidaan hyödyntää edistyneitä koneoppimis- ja syväoppimisratkaisuja. Näiden kirjastojen integroiminen vaatii yleensä kommunikointia Godotin ja ulkoisten kirjastojen välillä, koska Godot ei tarjoa niille suoraa tukea. (Godot Documentation, n.d.).

4.4 Tekniikan valinta

Tekoälytekniikan valinnassa on tärkeää ottaa huomioon pelin tarpeet ja vaatimukset. Tekniikan valinta riippuu monista tekijöistä, kuten pelin genrestä, pelimekaniikasta, suorituskyvystä ja halutuista ominaisuuksista. Yksinkertaisissa tai suorituskyvyiltään rajoitetuissa peleissä voidaan käyttää yksinkertaisia tekniikoita, kuten tilakoneita ja sääntöperusteisia järjestelmiä. Jos peliin tarvitaan reagoivia esimerkiksi NPC-hahmoja, kuten vihollisia, jotka reagoivat pelaajan liikkeisiin, tilakoneet ja polunetsintäalgoritmit voivat olla riittäviä. Monimutkaisemmissa peleissä, kuten strategia-, avoimen maailman ja simulaatiopeleissä, voidaan käyttää kuten käyttäytymispuita, monimutkaisempia tilakoneita ja päätöksentekojärjestelmiä. Jos peli vaatii tekoälyltä taktisia päätöksiä, kuten vihollisen liikkumisen tai hyökkäyksen pelaajaa vastaan, yksinkertaisemmissa järjestelmissä voidaan käyttää tähän myös tilakoneita. Korkeamman tason päätöksenteko esimerkiksi strategiapelissä, jossa NPC:t tekevät monivaiheisia valintoja pitkällä aikavälillä, voi taas edellyttää hieman monimutkaisempia lähestymistapoja, kuten käyttäytymispuita tai hyötypohjaisen tekoälyä. Jos taas pelissä on suuri avoin pelimaailma, polunetsintäalgoritmit (esimerkiksi A*) ja navigointiväylät voivat auttaa tekoälyä liikkumaan esteiden ympäri ja etsimään reittejä, kun taas yksinkertaisemmassa reitityksessä voidaan käyttää valmiin reitin seurantaa. (Millington 2000, 12–14.)

Monimutkaisempia algoritmeja, kuten syväoppimista ja neuroverkkoja, voidaan käyttää erityisesti pelin vaatiessa älykästä ja dynaamista käyttäytymistä. Tällaiset algoritmit voivat tuottaa vaikuttavaa tekoälyä, joka reagoi ympäristön muutoksiin ja pelaajan toimiin realistisesti. Kuitenkin nämä teknologiat voivat olla laskennallisesti raskaita ja vaikeita toteuttaa, joten pelin suorituskyky tai kehitysaikataulu voivat asettaa rajoituksia niiden käytölle. Vahvistusoppiminen (engl. *Reinforcement learning*) mahdollistaa tekoälyn kehittymisen peliympäristönsä toiminnasta kokeilemalla eri toimintoja ja oppimalla niiden seurauksista. Esimerkiksi itseajavat autot tai pelissä oppivat viholliset voivat kehittyä ajan myötä ja sopeutua ympäristön muutoksiin. Ohjattu oppiminen (engl. *Supervised learning*) taas sopii tilanteisiin, joissa tekoäly on koulutettu ennalta määritellyillä säännöillä ja esimerkeillä, kuten tiettyjen taktiikoiden tai käyttäytymismallien tunnistamisessa. (Millington 2000; Jagli, Chandra, Dhanikonda & Laxmi 2024.)

On tärkeää huomata, että algoritmien ja tekniikoiden valinnan lisäksi onnistuneiden ratkaisuiden löytäminen vaatii paljon muunlaisiakin valintoja ja ratkaisuja pelin suunnittelussa. Erityisesti valinta on yhteydessä pelin muuhun infrastruktuuriin. Lisäksi tekoälyn käyttö on liitoksissa moniin muihinkin osiin: hahmojen havainnointiin, animaatioihin, fysiikkaan ja ylipäätään kaikkeen tietoon, minkä perusteella

tekoälyn halutaan päätöksiä tekevän. Tekoälyn saaminen toimimaan halutuilla tavoilla voi vaatia myös paljon iterointia ja optimointia. (Millington 2000, 12.)

5 TEKOÄLYN KEHITTÄMINEN DEMOSOVELLUKSEEN

Tässä luvussa käsitellään ensin, millainen opinnäytetyön käytännön sovellutuksena toteutettu demopeli on ja miten se on rakennettu Godot-editorin työkaluilla. Sen jälkeen tarkastellaan, kuinka hyötypohjainen tekoäly voidaan rakentaa luomalla uusia noodeja ja laajentamalla niiden toimintoja skripteillä. Lopuksi pohditaan, millaiset tekijät vaikuttavat siihen, kuinka hyvin tekoäly voidaan rakentaa valitulla menetelmällä.

5.1 Suunnitelma demopelille

Demosovelluksena kehitettiin agenttityyppinen yksinpelattava 2D-peli, jossa ohjataan yksinkertaisin grafiikoin animoitua hahmoa. Pelin voidaan määritellä kuuluvan sivuttaisvieritettävien ammuntapelien genreen, jossa on myös hiiviskelyelementtejä. Pelissä liikutaan 2D-ympäristössä sekä horisontaalisesti että vertikaalisesti, ja pelaajan liikkuessa kentässä eteenpäin vastaan tulee vihollisia. Vihollisten liikkumisessa ja päätöksenteossa hyödynnetään tässä työssä aiemmin käsiteltyä hyötypohjaista tekoälyä.

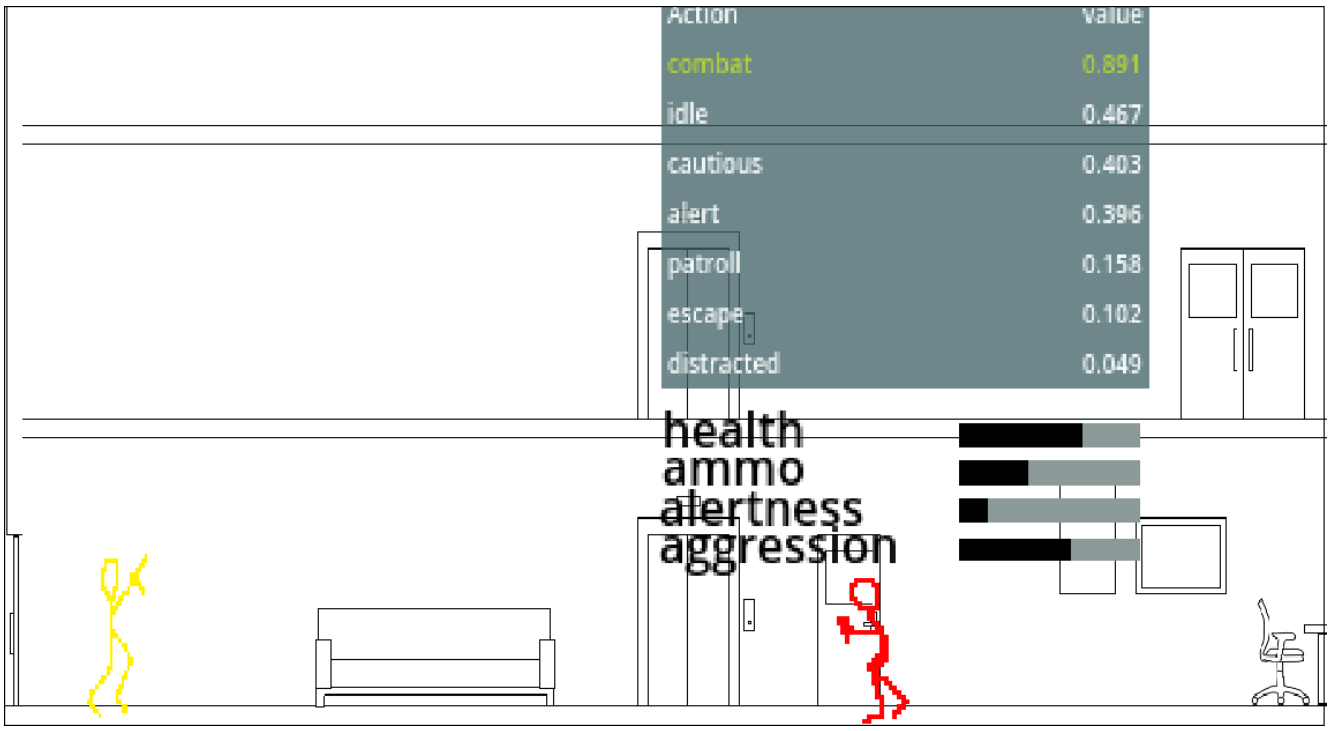
Peliin kehitettävän tekoälyn pohjana toimii tilakone, jonka toiminnalliset tilat jäljittelevät ihmisen toimintaa ja käyttäytymistä tietyssä tilanteessa. Tilakoneen lisäksi tekoälyn päätöksenteon luomiseksi käytetään hyötypohjaista tekoälytekniikkaa. Hyötypohjainen tekoäly tekee päätöksiä siitä, millainen käyttäytyminen on sopivaa kussakin tilanteessa. Käyttäytyminen voi olla mitä tahansa ennalta määriteltyä: suojautumista, ampumista tai vartiointia. Käyttäytymisen tai toimintojen pisteytys perustuu muuttujiin tai muuhun pelimaailmasta johdettuun dataan. Se, miten toiminnot pisteytetään, vaatii pohdintaa siitä, mitkä tekijät vaikuttavat tietyn toiminnon käynnistymiseen ja miten eri toimintojen pisteet suhteutetaan toisiinsa.

5.2 Peliympäristön luonti

Peliympäristönä käytetään 2D-kenttää, jossa pelaaja liikuttaa yksinkertaisesti toteutettua hahmoa tason päällä. Pelaaja voi siirtyä huoneesta toiseen, ja huoneissa on erilaisia objekteja, joiden kanssa pelaaja voi olla vuorovaikutuksessa. Lisäksi huoneissa on vihollishahmoja, jotka vartioivat rakennusta. Kentässä on myös yksinkertaisia piilopaikkoja ja esteitä.

Peliin rakennettiin yksinkertainen kenttä käyttämällä *Sprite*- ja *CollisionShape2D*-noodeja. *Sprite*-noodi on noodityyppi, jota käytetään 2D-kuvien tai animaatioiden näyttämiseen, ja sen avulla voidaan renderöidä kuvia näytölle. *CollisionShape2D*-noodia taas käytetään määrittämään 2D-objektin törmäysalue. Se ei itsessään näy pelissä, mutta mahdollistaa törmäysten havaitsemisen ja fysiikan laskemisen.

Pelaajan ohjaaman hahmon liikkumista ja näihin liittyviä animaatioita varten luotiin tilakone. Käytetty animaatio riippuu pelaajan painamista ohjausnapeista, mutta myös esimerkiksi pelin tilanteesta ja hahmon sijainnista tietyllä hetkellä. Tilakoneesta tehtiin laajennettava, jotta siihen voidaan helposti lisätä uusia tiloja. Tilakonetta rakentaessa on tärkeää huomioida, mihin kaikkiin tiloihin mistäkin tilasta on mahdollista siirtyä. Lisäksi pelaajalle luotiin yksinkertainen energiatasomittari. Lähtökohtana on, että sekä pelaajan että vastustajan täytyy osua toisiinsa useamman kerran ennen tuhoutumistaan. Pelaaja pystyy liikkumaan eteen ja taakse, hyppäämään ja menemään kyykkyyn. Tämän lisäksi pelaaja pystyy ampumaan eri suuntiin.



KUVA 5. Kuvakaappaus pelistä yksinkertaisilla grafiikoilla.

Demopelissä on vain yhdentyyppisiä NPC-vihollisvastustajia, ja hyötypohjaista tekoälyä testattiin tämän työn aikana vain yhdelle NPC:lle. Tekoälyn käyttöä on kuitenkin tarkoitus laajentaa myöhemmin. Myös NPC:lle luotiin tilakone, joka ohjaa hahmon käyttämää animaatiota ja suorittamaa toimintoa eri tilanteessa. Kuvassa 5 on kuvattu, miten vihollisvastustajan tilat ja osa ominaisuuksista on tuotu näkyväksi

peliin debuggausta varten. Lisäksi tuli toteuttaa näkökenttä, joka vaikuttaa siihen, havaitseeko NPC pelaajan. Tähän voitiin käyttää *RayCast2D*- tai *Collider*-noodeja. Se toimii NPC:n aistimena, jonka kautta se saa tietoa siitä, onko pelaaja lähettyvillä. Näkökentän pituutta, suuntaa ja suoritusaikaa voidaan säätää sen mukaan, mihin suuntaan NPC katsoo ja missä tilassa se on. NPC:llä on lisäksi pääsy muihin tietoihin siitä, mitä pelissä tapahtuu. Koska kyseessä on tietokonepeli eikä esimerkiksi oikeaa ympäristöä havainnoiva robotti, jonka tarvitsee tulkita videokuvaa ympäristöstään, tiedot voidaan jakaa suoraan koodista ilman, että niitä tarvitsee hankkia monimutkaisen datan analysoinnin kautta. Lisäksi tietoa voidaan tuottaa lisää. Esimerkiksi vastustajalla voi olla tieto siitä, kuinka monta laukausta on ammuttu, mikä on pelaajan energiataso tai vaikkapa kuinka avoimesti pelaaja on osuttavissa. Tekoälyn päätöksentekologiikan tehtävänä on määrittää, miten näihin tietoihin reagoidaan. NPC:llä on pelissä useita tiloja, jotka määrittävät miten ne voivat käyttäytyä kussakin tilanteessa ja siihen, mitä tietoja ne voivat käyttää. Pelin arkkitehtuuri ja koodi rakennettiin siten, että näitä tiloja voidaan helposti tarvittaessa lisätä.

TAULUKKO 1. Demopelissä kokeiltuja NPC:n tiloja, jotka määrittävät, miten se käyttäytyy kussakin tilanteessa.

Tila	**NPC:n käyttäytyminen**
Lepotila	Tässä tilassa ympäristöön ei kiinnitetä paljon huomiota, ja liikkuminen voi olla vähäistä. Tämä on yksinkertainen perustila, josta voidaan siirtyä muihin tiloihin.
Partiointi	Tässä tilassa NPC vartioi tiettyä aluetta valppaana, mutta ei ole tietoinen pelaajan olemassaolosta.
Taistelu	Tässä tilassa NPC:llä on näköhavainto pelaajasta, ja se yrittää eliminoida pelaajan. Tämä tila on monimutkaisempi, koska siinä täytyy ottaa huomioon monia tekijöitä, jotta NPC:n toiminta olisi realistista ja tilanteeseen sopivaa.
Häiritty	Tässä tilassa NPC on keskittynyt johonkin tekemiseen niin paljon, että hän ei havainnoi ympäristöä kunnolla.
Hälytys	Tässä tilassa NPC tietää, että jokin on pielessä, ja etsii aktiivisesti pelaajaa tai uhkaa. NPC saattaa myös kutsua muita apuun.
Jahtaus	NPC on tietoinen pelaajasta ja tämän sijainnista ja liikkuu pelaajaa kohti yrittäen estää mahdollisen pakenemisen. NPC saattaa mennä tähän tilaan, jos pelaaja lähtee pakoon.

(jatkuu)

TAULUKKO 1. (jatkuu)

Pakeneminen	Kun tilanne sitä vaatii, myös NPC voi tehdä päätöksen paeta. Tällöin se voi lähteä etääntymään pelaajasta tai piiloutua.
Epäileväinen	NPC on hieman varuillaan, mutta saattaa silti olla keskittynyt johonkin asiaan, eikä ole täysin valppaana. Liikkeet ovat kohtalaisen hitaita ja keskittyminen tiettyyn kohtaan kentässä on voimakasta.
Varovainen	NPC on valmistautunut reagoimaan ympäristön muutoksiin ja havainnoi ympäristöä, mutta ei tiedä, missä pelaaja on.

NPC voi olla pelissä vain yhdessä tilassa kerrallaan, mutta tila voi vaihtua nopeasti tilanteen muuttuessa. Jokaisen tilan sisällä hahmo voi suorittaa useita erilaisia toimintoja. Tekoälytoiminnot ohjaavat sitä, missä tilassa NPC kulloinkin on, mutta ne voivat lisäksi määrittää, miten hahmo käyttäytyy kyseisessä tilassa. Joissain tiloissa toiminta on yksinkertaista eikä vaadi monimutkaista ohjausta – esimerkiksi lepo- tai partiointitilassa vihollinen liikkuu pääasiassa ennalta määritellyllä alueella pelimaailmassa. Sen sijaan taistelutila vaatii monimutkaisempaa päätöksentekoa. Vihollisen tulisi ottaa huomioon pelaajan toiminta ja arvioida omia mahdollisuuksiaan onnistua tilanteessa. Jos pelaaja esimerkiksi lähestyy suoraan ampumaetäisyydelle eikä vihollisella ole muuta keinoa selviytyä, sen tulisi pyrkiä ampumaan pelaajaa sen sijaan, että se yrittäisi paeta. Toisaalta, jos NPC:n energiat ovat vähissä ja pakeneminen on mahdollista, olisi järkevää käyttää tämä vaihtoehto. Lisäksi vihollisen tulisi osata suojautua ja hyödyntää tilanteita, joissa se voi ampua pelaajaa turvallisesti. Tällaiset valinnat perustuvat paljolti myös pelimekaniikkaan ja pelin muuhun suunnitteluun, mutta hyötypohjainen tekoäly mahdollistaa tilanteeseen sopivan ja dynaamisen käyttäytymisen, tarvittaessa hyvin monipuolisin tavoin.

5.3 Hyötypohjaisen tekoälyn kehittäminen demopeliin

Godotissa pelin tekoälyn arkkitehtuurin rakentaminen voidaan toteuttaa useilla eri tavoilla. Valinta riippuu siitä, kuinka joustavasti tekoälyn rakennetta ja logiikkaa halutaan hallita, kuinka optimoitua ratkaisua haetaan, käyttötarkoituksesta (halutaanko tehdä työkalu esimerkiksi muille) ja pelin monimutkaisuudesta. Tässä työssä tekoälyn toteuttamiseen hyödynnettiin Godotin valmista noodijärjestelmää laajentaen sitä uusilla noodeilla ja skripteillä. Godotissa pelinteon käytännöt pohjautuvat usein oliokeskeisen suunnittelun periaatteiden soveltamiseen, ja pelin rakenne muodostuu pääasiassa skeneistä, noodeista ja skripteistä. Koska Godot on rakennettu näiden osien ympärille, niiden järkevä ja tehokas käyttö on keskeistä luotaessa pelejä, jotka ovat sekä helposti ylläpidettäviä että tehokkaita.

Näin voidaan käyttää komponenttien jakamiseen Godotin skenepuuta ja luoda räätälöityjä noodeja, joista voi koota tekoälyn toimintoja. (Godot documentation, n.d.)

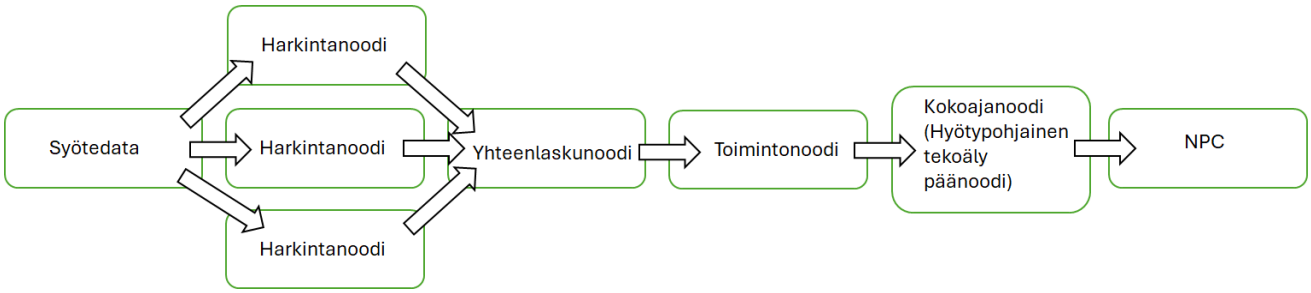
Erillisten noodien käyttö mahdollistaa pienempien, itsenäisten komponenttien rakentamisen, jotka suorittavat kukin tiettyjä tehtäviä. Tämä lähestymistapa tekee koodista helpommin ylläpidettävän ja laajennettavan, sillä yksittäisiä noodeja voidaan lisätä tai muokata ilman, että koko järjestelmä menee rikki. Samalla vältetään liian pitkän ja vaikeasti luettavan koodin kirjoittaminen yhteen skriptiin. Skriptit pyrittiin tässä työssä pitämään yksinkertaisina ja selkeinä niin, että ne sisälsivät vain hyötypohjaisen tekoälyn keskeisimmät perustoiminnot.

Ajatuksena oli kehittää noodeihin perustuva puurakenne, joka olisi mahdollisimman yksinkertainen ja laajennettavissa oleva pohja hyötypohjaiselle tekoälylle, jonka koodia pystyy jatkamaan ja kustomoimaan erilaisten pelien tarpeisiin. Toteuttamani työkalu tarjoaa täyden hallinnan kooditasolla ja yhdistää tähän visuaalisen käyttöliittymän skenepuun ja *Inspector*-näkymän avulla, mikä tekee sen käytöstä joustavaa ja selkeää. Työkalussa käytetään Godotin *Inspector*-näkymässä tarjoamia mahdollisuuksia muuttujien säätämisessä ja valmiiden noodien lisäämisessä. Työ tuo esille, kuinka kehittäjä voi Godotissa tehdä omia toteutuksia ja käyttää mitä tahansa valitsemiaan algoritmeja, jotenkehittäjällä on suhteellisen vapaat kädet toteuttaa visioitaan.

Godotin skenepuun rakentaminen tuo visuaalisen ulottuvuuden, mikä tekee tekoälyn rakenteen ymmärtämisestä ja laajentamisesta helpompaa. Sen avulla voi nähdä miten eri noodit liittyvät toisiinsa ja miten tiedonsiirto kulkee niiden välillä, mikä helpottaa myös virheiden etsintää ja testausta. Kun noodien tilaa voi tarkastella ja seurata visuaalisesti, debuggaus voi olla huomattavasti helpompaa. Muuttujien esittämiseen visuaalisesti Godotin *Inspector*-ikkunassa käytetään `@export`-komentosanaa. Se mahdollistaa arvojen säätämisen ilman, että niitä tarvitsee muuttaa skriptissä, mikä voi myös helpottaa kehittämistä, debuggausta ja tuoda ratkaisuun visuaalista hallintaa. Godotissa voitaisiin myös rakentaa kehitetyn ratkaisun ympärille laajemmin visuaalista käyttöliittymää. Godotissa on valmiina erilaisia luokkia, joilla kyettäisiin rakentamaan editorityökalu tai plugin (esimerkiksi editorplugin-luokka).

5.4 Noodien hierarkia ja yhteydet

Noodien hierarkia ja yhteydet Godotissa on kuvattu kuviossa 8. Kuviossa esitetyllä noodirakenteella voidaan kehittää hyötypohjainen tekoäly, joka on helposti laajennettavissa. Näin sen säätäminen on myös suhteellisen helppoa, ja se voidaan kustomoida erilaisiin käyttötarkoituksiin.



KUVIO 8. Hyötypohjaisen tekoälyn noodien rakenne

5.4.1 NPC-noodi

NPC-noodi laajentaa *CharacterBody2D*-noodia. Se on perusnoodi, jota käytetään 2D-pelihahmoja luotaessa. Kaikissa muissa noodeissa laajennetaan perusnoodiluokkaa, koska niissä ei tarvita muita valmiita ominaisuuksia. NPC-noodi on siis tekoälyhahmo, jota ohjataan hyötypohjaisella tekoälyllä. Noodi on erotettu erilliseksi skeneksi, ja tämän noodin alle voidaan lisätä lapsinoodeiksi kaikkea muuta mikä liittyy NPC-hahmoon – myös hyötypohjainen tekoäly kaikkine noodeineen. Tällä noodilla on oma skriptinsä, joka sisältää kaiken NPC:hen liittyvän koodin, kuten animaatiot, toiminnot ja muuttujia. Ne jaetaan hyötypohjaiselle tekoälylle käyttöön, jotta se voi käyttää niitä datana hyödyllisten toimintojen toteuttamisen arvioimiseen. Noodi kuuntelee signaalin avulla hyötypohjaisen tekoälyn antamaa tietoa toimintojen hyödyllisyysjärjestyksestä ja toteuttaa halutut toiminnot.

5.4.2 Kokoaja-noodi

Kokoaja-noodi on hyötypohjaisen tekoälyn päänoodi, jonka tehtävänä on käsitellä ja arvioida toimintoja niiden hyötypisteiden perusteella. Tämän noodin skriptissä lähetetään signaali korkeimman pistemäärän saaneen toiminnan vaihtumisesta, jota NPC:n skriptissä kuunnellaan. Tekoälyn toiminta perustuu `_physics_process(delta)`-funktioon, joka on Godotin `engine callback` -funktio. Tämä funktio kutsutaan automaattisesti joka ruudunpäivityksessä, ja se on suunniteltu käytettäväksi silloin, kun pelissä tarvitaan ajastettuja laskelmia, jotka liittyvät fysiikkaan, liikuttamiseen tai muihin peliin liittyviin laskentatehtäviin. Tämän vuoksi tekoälyä suoritetaan jatkuvasti pelin aikana. Skriptissä käydään läpi kaikki *action*-noodit, jotka on sijoitettu päänoodin alle. Näiden noodien pistemäärät verrataan

toisiinsa, ja jos korkeimpia pisteitä saanut toiminto muuttuu, siitä lähetetään signaali, jonka NPC-hahmon skripti kuuntelee ja reagoi. Tällä tavoin tekoäly pystyy jatkuvasti arvioimaan ja päivittämään toimintojaan pelin kulun mukaan.

5.4.3 Toimintonoodi

Toimintonoodit vastaavat toimintoja, joita NPC voi toteuttaa. Noodi ei sisällä toiminnon suorittamiskoodia, joka löytyy NPC:n skriptistä. Noodin tehtävänä on hallita tekoälyn toiminnon yksilöllistä tunnistetta ja toiminnon hyödyllisyysarvoa sen lapsisolmujen perusteella. Toimintonoodeja voidaan helposti lisätä. Tällöin NPC:llä tulee myös olla vastaavaan toimintoon liittyvät vastineet omassa koodissaan. Lisäksi toimintonoodilla tulee olla lapsina harkinta- ja yhteenlaskentanoodit (jos useampia harkintanoodeja), jotka esitellään seuraavaksi.

5.4.4 Harkintanoodi

Harkintanoodi toimii pohjana hyödyllisyysarvon laskemiselle. Noodin päätehtävänä on laskea ja palauttaa arviointipiste, joka kuvaa yksittäisen pelin muuttujan hyödyllisyyttä toiminnolle. Muuttuja voi olla esimerkiksi pelihahmon terveydentila, etäisyys kohteeseen, nopeus tai muu pelimaailman tieto. Tämä toteutetaan soveltamalla muokattavaa käyrää pisteisiin. Jokaisella toiminnolla voi olla useita harkintanoodeja, riippuen siitä mitä kaikkea muuttujia ja tietoa halutaan ottaa huomioon toiminnon hyödyllisyyden arvioinnissa. Arvot tulee skaalata samalle mitta-asteikolle, jotta niitä voidaan verrata, minkä vuoksi skaalaus on hyvä myös toteuttaa tässä noodissa. Monesti luku skaalataan desimaalilukuna välille 0,0–1,0. Harkintanoodi toimii siis pohjana mukautettujen arviointien luomiselle. Noodissa voidaan määrittää, mitä ja miten tiettyä pelin muuttujaa tai ominaisuutta käytetään arvioinnin laskemiseksi. Kokonaisuus vaatii kehittäjältä pohdintaa miten pelin tietoja tulisi käyttää, jotta tekoälyn toiminta olisi halutunlaista.

5.4.5 Yhteenlaskentanoodi

Jotta eri toimintojen lopullisissa pisteissä voidaan ottaa huomioon monenlaista dataa ja muuttujia, on yhteenlaskentanoodin tehtävänä suorittaa laskentaa useiden harkintanoodien kesken. Tämän noodin avulla voidaan tuottaa yksi lukuarvo, joka kuvaa kokonaisuudessaan yhden toiminnon hyödyllisyyttä. Tämän noodin avulla voidaan rakentaa monimutkaisempia arviointijärjestelmiä, joissa useiden tekijöi-

den painoarvot voidaan laskea yhteen halutulla tavalla. Noodiin lisätään funktioina useita erilaisia laskentavaihtoehtoja kuten summaus, kertominen, maksimi- ja minimiarvo sekä keskiarvo, joista voidaan valita tilanteeseen sopivin.

5.5 Skriptit

Jokaisella toteutuksen noodilla on oma skriptinsä, joka määrittelee sen toimintaa. Skriptit laajentavat Node-luokkaa, joka on Godotissa perusluokka, jota yleensä käytetään, kun ei tarvita valmiina erityisempiä toimintoja. NPC on yhteydessä tekoälyyn signaalilla. Tekoälyn noodit ovat yhteydessä toisiinsa funktioiden kutsuilla ja yhteisillä metodeilla.

Aivan kuten noodien määrän ja käytön kohdalla, myös skriptien kanssa kehittäjän tulee pohtia niiden käyttötapa esimerkiksi selkeyden, toimivuuden, laajentamismahdollisuuksien tai optimoinnin kannalta. Demopelin tekoäly on jaettu osiin noodirakenteella. Jokaisella noodilla on omat toimintonsa, ne ovat hierarkkisesti järjestetty ja ne jakavat tietoa toisilleen. Tässä luvussa käydään läpi, kuinka Godotin *GdScript*-kieltä käyttäen koodin hyötypohjainen tekoäly rakennettiin toimivaksi.

5.5.1 NPC-skripti

NPC-skripti laajentaa *CharacterBody2D*-noodia, joka on yleisesti hahmojen luomisessa käytetty noodi. Se sisältää sisäänrakennettuja toimintoja, kuten liikkeen ja törmäyksien hallintaa. NPC-hahmoskriptissä määritellään hahmon ominaisuuksia ja muuttuvia arvoja. Niitä voidaan keksiä paljon lisää, mutta tässä työssä pysytään yksinkertaisissa perusmuuttujissa, jotka esitellään kuvassa 5.

```
# Exported variables that can be adjusted in the Godot editor
@export var health = 100
@export var ammo = 100
@export var alertness = 0
@export var aggression = 0
@export var detection_range = 0
@export var shoot_range = 0

# Action states
var is_fighting = false
var is_searching_player = false
var is_in_cover = false
var is_retreating = false
var is_patrolling = false
var is_reloading = false
```

KUVA 5. Määriteltyjä muuttujia, jotka on tuotu näkyviksi Godot-editorissa.

Kuvan 5 mukaisesti pelissä määritettiin muuttujia, joita tekoäly käyttää päätöksenteossa. Godotissa `@export`-avainsana mahdollistaa muuttujien näkyvyyden ja muokkaamisen suoraan Godot-editorissa. Boolean-tyyppisillä muuttujilla voidaan tarkistaa, onko hahmo juuri tekemässä jotain tiettyä toimintoa. `Current_target`-muuttujalla voidaan määrittää hahmolle kohdeobjekti.

NPC-hahmo kuuntelee signaalia, jonka UtilityAI-agentti sille lähettää. Silloin käytännössä ajetaan funktio `on_utility_ai_agent_tio_score_action_changed`, jonka parametrina on korkeimmat pisteet saaneen toiminnon nimi. Tämän jälkeen suoritetaan vastaava toiminto, joka vastaavat kaikesta siitä toiminnasta, mitä hahmo tekee ollessaan tietyssä tilassa (KUVA 6).

```
82  func _on_utility_ai_agent_top_score_action_changed(top_action_id):
83      print("Action changed: %s" % top_action_id)
84
85      match top_action_id:
86          "idle":
87              idle()
88          "reload":
89              reload()
90          "patroll":
91              patroll()
```

KUVA 6. NPC:n signaalifunktio.

Signaalin lähettäessä tiedon korkeimmat pisteet saaneesta toiminnosta, vastaavaa funktiokutsu suoritetaan. Tämän lisäksi skripti sisältää siis myös funktiot, joissa on implementoitu itse toiminto. NPC:llä on erilaisia jatkuvasti muuttuvia muuttujia kuten terveys, tarkkaavaisuus, aggressiivisuustaso, panosten määrä jne. Kaikki nämä muuttujat toimivat eri tavoin, ja niille tarvitaan myös siis funktiot, jotka hallitsevat jokaisen muuttujan toimintaa ja muutoksia. Tähän käytetään Godotin valmista `_process(delta)`-funktiota. Tämä funktio kutsutaan automaattisesti joka ruudunpäivityksessä (*frame*), ja se saa parametriksi `delta`, joka kertoo, kuinka paljon aikaa on kulunut edellisestä kehyshetkestä. Tämä on koodissa kohta, joka pelimoottorissa laittaa tekoälyn pyörimään ja mahdollistaa sen, että pelissä voidaan päivittää kaikki jatkuvasti muuttuvat arvot.

5.5.2 UtilityAI

UtilityAi on tekoälyn päänoodi, jonka tehtävänä on ajaa hyötypohjaista tekoälyä, koota pisteet lapsina olevilta toimintonoodeilta ja lähettää signaali (KUVA 7). UtilityAI -skriptissä laajennetaan noodiluokan toiminnallisuutta, kuten muillakin skripteillä. Tämän lisäksi keskeinen tehtävä on lähettää signaali, jota NPC kuuntelee. Signaali lähetetään, kun eniten pisteitä saanut toiminto muuttuu. Lisäksi UtilityAI:n skriptissä kootaan kaikki pisteet ja tallennetaan ne muuttujiin. Funktio `evaluate_actions` käy läpi kaikki lapsinoodit (jotka ovat *UtilityAiAction*-tyyppisiä noodeja), ja hakee korkeimmat pisteet saavan toiminnon. Jos paras toiminto on muuttunut edelliseen verrattuna, funktio lähettää signaalin `highest_utility_action_changed` ja päivittää `current_highest_utility_action`-muuttujan arvon. UtilityAI-noodissa ei itse lasketa pisteitä, vaan kutsutaan kunkin toiminnon omaa `calculate_score`-metodia, jolla pisteet määritetään.

```
4   class_name UtilityAi extends Node
5
6   # This signal is emitted when the highest-scoring action changes
7   # The npc listens to this signal and reacts with its own behavior logic
8   signal highest_utility_action_changed(highest_utility_action_id)
9
10  # Enable or disable the AI
11  @export var is_utility_AI_active: bool = true
12
13  # Variables for tracking and evaluating action utilities in AI decision-making
14  var current_highest_utility_action
15  var action_scores = []
16  var are_scores_sorted = false
17  var all_utility_scores = []
```

KUVA 7. Tekoälyn tilaa ja parhaan toiminnon seurantaa varten määritellyt muuttujat ja signaali.

5.5.3 UtilityAIAction

UtilityAiAction-skripti määrittelee luokan, joka edustaa yksittäistä tekoälyn toimintoa, kuten hyökkäämistä tai pakenemista (KUVA 8). Luokan noodin nimi tallennetaan toiminnon tunnisteeksi omaan muuttujaansa. Toiminnon hyötyarvo haetaan sen lapsinoodeilta. Mikäli lapsinoodeja ei ole, hyötyarvoksi palautetaan nolla, jolloin toimintoa ei pidetä tekoälyn näkökulmasta kannattavana. Lapsinoodeina käytetään *consideration*-luokkaa ja *aggregation*-luokkia, jotka arvioivat tilanteeseen vaikuttavia tekijöitä ja palauttavat niiden perusteella hyötypisteytyksen. Vaikka skripti itsessään ei sisällä muita toimintoja, sen toteuttaminen omana luokkanaan ja noodinaan on rakenteellisesti perusteltua. Se selkeyttää tekoälyn rakennetta sekä helpottaa sen kehittämistä ja laajentamista jatkossa.

```
3
4   class_name UtilityAiAction extends Node
5
6   # Action identifier, defaults to the node's name
7   var action_id = self.name
8
9   # Function that gets the utility score from the child node
10  func get_score_from_child() -> float:
11      var considerations = self.get_children()
12
13      if considerations.size() == 0:
14          return 0.0
15
16      var consideration = considerations[0]
17      return consideration.get_score_from_child()
```

KUVA 8. Toiminto-noodin skripti, joka hakee hyötyarvon lapsinoodilta.

5.5.4 UtilityAIAggregation

UtilityAiAggregation-skripti tarjoaa mekanismeja monien *consideration*- sekä *aggregation*-noodien (arviointikriteerien) yhdistämiseen, joista voidaan laskea yksi toiminnon hyödyllisyyttä kuvaava kokonaisarvo (*score*). *Aggregation*-noodi asetetaan *action*-noodin lapsinoodiksi. Sen tehtävänä on laskea yhteinen pistemäärä useista lapsinoodeista käyttäen valittua yhdistämistapaa (*aggregation method*). Skriptissä määritellään useita tapoja yhdistää pisteitä useista *aggregation*- ja *consideraton*-noodeista (KUVA 9). Yleinen *consideration*-noodin tapa laskea pisteet yhteen kertolaskun käyttäminen, mutta pisteet voidaan yhdistää muillakin tavoin, riippuen miten pelin muuttujien halutaan vaikuttavan *action*-noodin yhteispisteisiin. (KUVA 10.) Esimerkiksi voidaan päättää, että jokin *boolean*-tyyppinen muuttuja on niin tärkeä toiminnolle, että sen ehtojen toteutuessa (ollessa *true*) nostetaan toiminnon hyötyarvo heti maksimiarvoon. Skriptissä määritellään funktiot jokaiselle eri laskutavalle ja niissä funktioissa käydään *consideration*-noodit läpi yhdistäen pisteet omilla tavoillaan. Käyttäjä voi siis *aggregation*-noodien avulla valita, miten *consideration*-noodien pisteet lasketaan yhteen. *Action*-noodin alle voidaan lisätä useita *aggregation*- ja *consideration*-noodeja. Käyttäen näitä kahta noodia voidaan jokaisen toiminnon pisteet laskea yksilöllisellä tavalla ja ottaa huomioon erilaisia muuttujia. Jokainen toimintonoodi saa kuitenkin lopulta vain yhden hyödyllisyyttä kuvaavan arvon, joka on luotu lisäämällä *aggregation* ja *consideration*-noodeja halutulla tavalla.

Ainoastaan UtilityAiAgent on yhdistetty NPC-hahmon noodiin signaalin avulla. UtilityAiAgentin eri noodit on yhdistetty funktion kutsuilla, kuten `get_score_from_child`-funktiolla. Godotissa voi helposti viitata toisien noodien koodiskriptiin viittaamalla noodin sijaintiin skenepuussa. Kutsumme siis lapsinoodien funktioita ja nämä palauttavat pisteet ylöspäin skenepuussa.

```
2   class_name UtilityAiAggregation extends Node
3
4   # Defines an enum type for different aggregation methods
5   enum AggregationMethods {
6       MULTIPLY,
7       SUM,
8       AVERAGE,
9       MAXIMUM,
10      MINIMUM
11  }
12  # Defines the aggregation_type variable, which can be selected in inspector
13  @export var aggregation_method: AggregationMethods
14
15  # Maps each aggregation method to its corresponding function
16  var aggregation_map := {
17      AggregationMethods.MULTIPLY: self.multiply,
18      AggregationMethods.SUM: self.sum,
19      AggregationMethods.AVERAGE: self.average,
20      AggregationMethods.MAXIMUM: self.maximum,
21      AggregationMethods.MINIMUM: self.minimum,
22  }
```

KUVA 9. Yhteenlaskentamenetelmien liittäminen vastaaviin funktioihin.

```
39
40    # Multiplies the scores of all child nodes and returns the result
41    func multiply() -> float:
42        var scores = 1.0
43        var number_of_considerations = 0
44
45        for child in get_children():
46            if is_acceptable_child(child):
47                scores *= child.get_score_from_child()
48                number_of_considerations += 1
49
50        return scores if number_of_considerations > 0 else 0.0
51
```

KUVA 10. Esimerkki multiply-funktiosta.

5.5.5 UtilityAIConsideration

UtilityAIConsideration-noodin tehtävänä on laskea hyötyarvoja (*utility scores*) NPC:n tai minkä tahansa pelin muuttujien, ominaisuuksien tai metodien perusteella. Noodissa käytetään Godotista löytyvää valmista työkalua: *Curve*-objektia. Sen avulla voidaan määrittää, miten haluttu muuttuja pisteytetään sen arvon muuttuessa. *Curve* toimii visuaalisena työkaluna, jossa X-akseli edustaa muuttujan arvoa ja Y-akseli sen vastaavaa hyötyarvoa. Näin voidaan joustavasti säätää, kuinka paljon tiettyjen arvojen suuruudet vaikuttavat toiminnon valintaan (KUVA 11).

KUVA 11. *Curve*-objekti Godotin *Inspector*-näkymässä.

Kuvan 12 skriptissä alustetaan valmiit paikat noodeille ja niiden muuttujille, joihin voidaan myöhemmin asettaa *Inspector*-näkymässä haettavan haluttu arvo pelistä viittaamalla noodiin ja sen muuttujaan. Tämän jälkeen haetaan X:ää vastaava Y:n arvo käyttäen *Curve*-luokan `sample_baked`-metodia. Näin saadaan laskettua hyötyarvo jokaiselle yksittäiselle muuttujalla. (KUVA 13.)

```
2   """ A consideration evaluates a specific variable or method and returns a score """
3
4   class_name UtilityAiConsideration extends Node
5
6   # This variable refers to the node from which we want to retrieve a value
7   @export var node: Node
8
9   # This is the name of the property or method from which the value will be retrieved
10  @export var property_name: String = "";
11
12  # This defines the propertys maximum value used as the scaling reference
13  @export var max_value: float = 1.0;
```

KUVA 12. Muuttujat voidaan asettaa editorissa *@export*-attribuutin avulla.

```
20  # Calculates the final score by applying a curve to the raw score
21  func get_score_from_child() -> float:
22      return apply_curve(score())
23
24  # Applies the curve to the raw score, or returns 0 if no curve is defined
25  func apply_curve(score: float) -> float:
26      if curve == null:
27          return 0.0
28      return curve.sample_baked(score)
29
```

KUVA 13. Alkuarvon muuttaminen käyrän avulla lopulliseksi hyötyarvoksi.

5.6 Huomioita hyötypohjaisen tekoälyn toteutuksesta

Hyötypohjainen tekoäly -malli osoittautui käytännössä joustavaksi ja helposti laajennettavaksi lähestymistavaksi. Godot-pelimoottorin skenepuu ja mahdollisuus tuoda muuttujia *Inspector*-näkymään toivat visuaalisuutta ja selkeyttä. Kustomoidut noodit mahdollistivat tarkemman hallinnan: yksittäisten tekoälykomponenttien toimintaa saattoi säätää vaikuttamatta muuhun koodiin. Tämä teki järjestelmästä helposti ylläpidettävän ja muokattavan esimerkiksi silloin, kun haluttiin lisätä uusia käyttäytymismalleja tai toimintoja.

Hyötypohjaisen tekoälyn perusidea on lopulta melko yksinkertainen: tekoälylle määritellään joukko käyttäytymisiä, joille annetaan tilanteen mukaan muuttuvia pisteitä. Käyttäytymisten pisteytys perustuu pelidataan, kuten energiatilanteeseen, resurssien määrään tai etäisyyteen kohteista. Näitä muuttujia arvioidaan hyötyarvokäyrien (*curve*) avulla, jotka määrittävät, miten muuttujan arvo vaikuttaa tekoälyn päätöksiin. Pisteytysprosessi mahdollistaa tekoälyn sopeutumisen pelin kulkuun ja tilanteisiin. Käyttäytymisiä ei tarvitse aina valita puhtaasti pisteiden perusteella, vaan kehittäjä voi sisällyttää logiikkaan myös satunnaisuutta tai painotuksia. Tämä tekee järjestelmästä hyvin muokattavan: niin käyttäytymisten sisältö, valintaperusteet kuin reaktiotkin ovat kehittäjän määriteltävissä.

Haasteena tekoälyn toteutuksessa on määrittää, mitä kaikkea dataa tekoäly ottaa huomioon ja miten tämä tieto pisteytetään. Lisäksi ei aina ole selvää, miten eri käyttäytymisvaihtoehdot ovat suhteessa toisiinsa tai miten niiden prioriteetit tulisi määritellä. Tietyt muuttujat saattavat vaikuttaa useisiin käyttäytymisiin samanaikaisesti, jolloin voi syntyä tilanteita, joissa tekoälyn valinnat eivät vastaa odotettua lopputulosta. Tämä vaatii iteratiivista säätöä ja testaamista, jotta tekoälyn toiminta tuntuu loogiselta ja pelin kannalta mielekkäältä.

Toinen haaste liittyy pisteytyslogiikan hienosäätöön. Esimerkiksi *curve*-käyrien muotoilu voi vaikuttaa merkittävästi siihen, milloin tietyn muuttujan merkitys kasvaa tai vähenee. Jos käyrät ovat liian jyrkkiä tai epätasaisia, tekoäly voi tehdä yllättäviä tai epätoivottuja valintoja. Tämän vuoksi pisteytyksen taustalla olevan logiikan on oltava huolellisesti suunniteltu, ja sitä tulisi testata useissa pelitilanteissa.

Vaikka tässä työssä näin ei vielä tapahtunut, järjestelmän kasvaessa pisteytettävien muuttujien ja käyttäytymisten määrä voi johtaa siihen, että kokonaisuuden hallinta käy haastavaksi. Tällöin on tärkeää pitää rakenne modulaarisena ja dokumentoituna, jotta tekoäly pysyy ylläpidettävänä myös kehityksen

edetessä. Hyötypohjaisen tekoälyn suurin etu – sen joustavuus – voi kääntyä taakaksi, jos pisteytysjärjestelmästä tulee liian monimutkainen ilman selkeää rakennetta.

6 ARVIOINTIA JA JATKOKEHITYSTÄ

Tiedonhaku tekoälyn käytöstä Godotissa ja opinnäytetyön aikana toteutettu tekoäly olivat mielenkiintoinen projekti. Hyötypohjainen tekoäly osoittautui suorituskykyiseksi ratkaisuksi, jonka pohjarakenne on selkeä ja suhteellisen helppo toteuttaa. Koska järjestelmä on kevyt, se soveltuu hyvin myös mobiilipeleihin ja muihin resurssitehokkaisiin sovelluksiin. Toimintoja ja hyötyarvoja palauttavia noodeja voidaan lisätä joustavasti, mikä tekee tekoälyn suunnittelusta ja laajentamisesta varsin suoraviivaista. Lisäksi sitä voidaan rajoittaa tarpeen mukaan tai ottaa väliaikaisesti pois käytöstä. Visualisointi pelin sisällä tukee vahvasti tekoälyn testaamista ja kehittämistä. Esimerkiksi muuttujien hyötyarvot on hyvä näyttää pelinäkymässä reaaliaikaisesti, mikä helpottaa järjestelmän säätöä ja virheiden tunnistamista. Hyötyarvot on hyvä lajitella suuruusjärjestykseen, jolloin tekoälyn päätöksenteon logiikka tulee selkeämmin esille. Visualisointiin voidaan hyödyntää Godotin valmiita käyttöliittymäkomponentteja, kuten *ProgressBar*- ja *Label*-noodeja, joiden avulla muuttujien tilaa voi havainnollistaa pelaajalle tai kehittäjälle testausvaiheessa.

Demopelin kehityksen aikana vahvistui näkemys, että tekoälyn vaikutus pelaajakokemukseen ei riipu ainoastaan valitusta algoritmista, vaan myös sen soveltamisen laadusta sekä muista pelin rakenteellisista ratkaisuista. Koska tämän työn demopelin toiminnallisuudet olivat rajalliset, olisi tekoälyn voinut toteuttaa myös yksinkertaisella tilakoneella. Monet tässä työssä esitellyt tekniikat perustuvat kuitenkin samaan perusajatukseen: tekoäly valitsee tilanteeseen sopivan toiminnon, vaikka toteutustavat vaihtelevat. Eri algoritmeilla on omat vahvuutensa ja heikkoutensa, ja niiden soveltuvuus riippuu projektin tarpeista. Ratkaisun onnistuminen ei siis perustu pelkästään valittuun menetelmään, vaan siihen, kuinka huolellisesti ja tarkoituksenmukaisesti se toteutetaan. Pelaajakokemukseen vaikuttaa myös se, kuinka kattavasti peli reagoi erilaisiin tilanteisiin. Projektin aikana korostui lisäksi objektiorientoituneen kehitysmallin ymmärtämisen merkitys, sillä se parantaa koodin hallittavuutta, laajennettavuutta ja modulaarisuutta – erityisesti suuremmissa projekteissa.

Jatkossa tekoälyä on tarkoitus laajentaa lisäämällä NPC-hahmoille uusia toimintoja ja kehittämällä peliä monimutkaisemmaksi. Koodiin voidaan sisällyttää myös käyttöä tukevia ominaisuuksia, kuten virheenhavaitsemista ja tekoälyn toiminnan säätömahdollisuuksia. Projektin edetessä kävi selväksi, että tekoälyn toteutus on aina vahvasti sidoksissa siihen peliin, jossa sitä käytetään – sekä pelimekaniikkaan että ohjattaviin toimintoihin. Tässä korostuu yksilöllisten ratkaisujen merkitys.

Työ selkeytti tekoälyn hyödyntämistä Godotissa ja tarjosi vastauksia sille asetettuihin tutkimuskysymyksiin ja tavoitteisiin. Koin oppineeni paljon Godotilla kehittämisestä: miten sen perusrakenteita, kuten noodeja, voidaan hyödyntää kehityksen rakennuspalikkoina, miten kehitystyössä voidaan huomioida modulaarisuus ja skaalautuvuus sekä miten algoritmeista siirrytään luontevasti käytännön toteutukseen.

LÄHTEET

AI And Games. 2025. *AI 101: Introducing Utility AI*. Saatavissa: https://www.aiandgames.com/p/ai-101-introducing-utility-ai. Viitattu 10.2.2025.

Bedingfield, W. 2023. *Generative AI in Games Will Create a Copyright Crisis.* Wired. WWW-dokumentti. Saatavissa: https://www.wired.com/story/video-games-ai-copyright/. Viitattu 19.4.2025.

Brown, M. 2023. *The Genius AI Behind The Sims.* WWW-dokumentti. Saatavissa: https://gmtk.substack.com/p/the-genius-ai-behind-the-sims. Viitattu 29.01.2025.

Calvin, A. 2020. *CD Projekt used AI to lip-sync 10 languages in Cyberpunk 2077*. PC Games Insider. WWW-dokumentti. Saatavissa: https://www.pcgamesinsider.biz/news/71631/cd-projekt-used-ai-to-lip-sync-10-languages-in-cyberpunk-2077. Viitattu 15.4.2025

Chaudhari, V. 2017. *Goal Oriented Action Planning*. Medium. WWW-dokumentti. Saatavissa: https://medium.com/%40vedantchaudhari/goal-oriented-action-planning-34035ed40d0b. Viitattu 2.4.2025.

Endless Existence. 2024. *Behavior Tree for AI in Games: A Comprehensive Guide for Unity Developers*. WWW-dokumentti. Saatavissa: https://endlessexistence.com/2024/09/30/behavior-tree-for-ai-in-games-a-comprehensive-guide-for-unity-developers/. Viitattu 20.4.2025.

Fink, F. 2023. *Neural Networks and Gaming: How A.I. is Revolutionizing Game Design.* Medium. WWW-dokumentti. Saatavissa: https://medium.com/@franziska.fink.rpllc/neural-networks-and-gaming-how-a-i-is-revolutionizing-game-design-a28154ecf192. Viitattu 19.4.2025.

GameFromScratch. 2024. *Game Engine Popularity in 2024.* Saatavissa: https://gamefromscratch.com/game-engine-popularity-in-2024/. Viitattu 15.1.2025.

Godot Documentation. n.d. *Godot*. Saatavissa: https://docs.godotengine.org/en/stable/ Viitattu 10.1.2025.

Godot Engine. n.d. *godotengine/godot*. GitHub. Saatavissa: https://github.com/godotengine/godot Viitattu 15.4.2025.

Graham, D. 2013. *Game AI Pro: Collected Wisdom of AI Professionals*. Toim. Rabin, S. Boca Raton: CRC Press

Hendricks, T. 2023. *Don't Act, Behave: Performant Neural Networks in Game AI.* Game Developer. WWW-dokumentti. Saatavissa: https://www.gamedeveloper.com/design/don-t-act-behave-performant-neural-networks-in-game-ai. Viitattu: 19.04.2025.

History-Computer. 2024. *OXO Game Guide: History, Origin, and More.* WWW-dokumentti. Saatavissa: https://history-computer.com/technology/gaming/oxo-game-guide/. Viitattu 10.2.2025.

Jagli, D., Chandra, S., Dhanikonda, S.R. & Laxmi, N., 2024. Artificial Intelligence Usage in Game Development. PDF-dokumentti. Saatavissa: https://www.researchgate.net/publication/381931183_Artificial_Intelligence_Usage_in_Game_Development.Viitattu 25.3.2025

Kirby, N. 2011. *Introduction to game AI.* Boston: Course Technology.

Lague S. 2014. *A* Pathfinding.* YouTube. Saatavissa: https://www.youtube.com/watch?v=-L-WgKMFuhE. Viitattu 25.4.2025

Liat, C. 2015. *DeepMind's AI is an Atari gaming pro now.* Wired. WWW-dokumentti. Saatavissa: https://www.wired.com/story/google-deepmind-atari/. Viitattu 15.4.2025.

Linietsky. J. 2022. *Godot 4.0 will discontinue VisualScript.* WWW-dokumentti. Saatavissa: https://godotengine.org/article/godot-4-will-discontinue-visual-scripting/. Viitattu 25.01.2025.

Mescarenhas, H. 2017. *Grand Theft Auto 5 being used as simulation to test AI of driverless cars*. International Business Times UK. WWW-dokumentti. Saatavissa: https://www.ibtimes.co.uk/grand-theft-auto-5-being-used-simulation-test-ai-driverless-cars-1617438. Viitattu 15.4.2025.

Millington, I. 2019. *AI for games.* Boca Raton: CRC press.

Nnoli, I. 2023. *Generative AI Sparks Life into Virtual Characters with NVIDIA ACE for Games.* NVIDIA Developer. WWW-dokumentti. Saatavissa: https://developer.nvidia.com/blog/generative-ai-sparks-life-into-virtual-characters-with-ace-for-games?. Viitattu 15.4.2025.

Northwood, A. 2023. *The Rise of AI in Video Games: How Artificial Intelligence is Changing the Gam-ing Experience.* Medium. WWW-dokumentti. Saatavissa: https://medium.com/@alexnorthwood/the-rise-of-ai-in-video-games-how-artificial-intelligence-is-changing-the-gaming-experience-93e621df276a. Viitattu 10.3.2025.

Rabin, S. 2002. *AI game programming wisdom*. Massachusetts: Charles River Media.

Rasmussen, J. 2016. *Are behaviour trees a thing of the past? Game Developer.* WWW-dokumentti. Saatavissa: https://www.gamedeveloper.com/programming/are-behavior-trees-a-thing-of-the-past-. Viitattu 28.1.2025.

Read, D. 2004. *Utility theory from Jeremy Bentham to Daniel Kahneman.* Department of Operational Research, London School of Economics. PDF-dokumentti. Saatavissa: https://eprints.lse.ac.uk/22750/1/04064.pdf. Viitattu 28.1.2025.

Roberts, P. 2023. *Artificial intelligence in games.* Boca Raton: CRC Press.

Russell, S. & Norvig, P. 2003. *Artificial intelligence: a modern approach.* 3. painos. New Jersey: Pearson Education.

Schwab, B. 2009. *AI game engine programming.* 2. painos. Boston: Course Technology.

Sizer, B. 2018. *The Total Beginner's Guide to Game AI.* WWW-dokumentti. Saatavissa: https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/. Viitattu 29.01.2025.

Thompson, T. 2020. *Building the AI of F.E.A.R. with Goal Oriented Action Planning.* WWW-dokumentti. Saatavissa: https://www.gamedeveloper.com/design/building-the-ai-of-f-e-a-r-with-goal-oriented-action-planning. Viitattu 29.01.2025.

Toxigon Infinite. 2025. *Comparing Utility AI and Goal-Oriented Action Planning: Which is Right for You?* WWW-dokumentti. Saatavissa: https://toxigon.com/comparing-utility-AI-and-goal-oriented-action-planning. Viitattu 15.4.2025.

Yannakakis, G. N. & Togelius, J. 2018. *Artificial Intelligence and Games*. New York: Springer.