View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Simo Riikonen

PELAAJASTA KERÄTYN TIEDON KÄYTTÄMINEN PELIN VAIKEUSTASON NOSTAMISESSA

Opinnäytetyö
Toukokuu 2013

OPINNÄYTETYÖ
Toukokuu 2014
Tietojenkäsittelyn koulutusohjelma

Karjalankatu 3
80200 JOENSUU
p.013 260 600

Tekijä(t)
Simo Riikonen

Nimeke
Pelaajasta kerätyn tiedon käyttäminen pelin vaikeustason nostamisessa

Toimeksiantaja
-

Tiivistelmä

Opinnäytetyöni tutkii dynaamisen vaikeustason säätelyn sovellutuksia peleissä. Opinnäytetyössä vastataan seuraaviin kysymyksiin: millainen dynaaminen vaikeustason säätely on toiminut, millainen ei ole toiminut ja miten dynaamista vaikeustason säätelyä tulisi käyttää. Opinnäytetyön teoriaosuudessa perehdytään dynaamisen vaikeustason hyviin ja huonoihin puoliin. Teoriaosuudessa esitellään esimerkkejä, joissa dynaamisessa vaikeustason säätelyssä on onnistuttu tai ei ole onnistuttu. Lisäksi opinnäytetyössä on lyhyt katsaus proseduraaliseen sisällön generointiin, jota käytetään vaihtelun luomiseen peliprototyypissä.

Opinnäytetyön käytännön osuudessa toteutettiin 2D-ammuskelupeli, jossa pelaaja käy kaksintaisteluita tekoälyvastustajaa vastaan proseduraalisesti generoiduissa kentissä. Tekoälyvastustaja säätää omaa alustaan jokaisen tappion jälkeen pelaajasta keräämänsä tiedon perusteella. Teoriaosuuden tutkimuksen pohjalta peliprototyypin alustaksi valittiin Unity3D-pelimoottori.

Opinnäytetyön tuloksena syntyi peliprototyyppi, joka kerää tietoa pelaajasta ja luo kerättyjen tietojen pohjalta uusia vihollisia.

Kieli	Sivuja
suomi	51

Asiasanat
dynaaminen vaikeustason säätely, proseduraalinen sisällön generointi, Unity3D, pelinkehitys

THESIS
May 2014
Degree Programme in Business Information Technology
Karjalankatu 3

80200 JOENSUU
FINLAND
p.013 260 600

Author(s)
Simo Riikonen

Title
Using Information Collected from the Player to Increase the Difficulty of the Game

Commissioned by
-

Abstract

This thesis studies dynamic difficulty adjustment, and how it has been used in video games. This thesis answers the following research questions: what kind of dynamic difficulty adjustment works, what does not work, and how should dynamic difficulty adjustment be used. The theoretical part of the thesis examines the advantages and disadvantages of dynamic difficulty adjustment. The theoretical part contains examples of games that have either succeeded or failed in the creation of the dynamic difficulty. Additionally, a brief look into procedural content creation is taken. It is used to create variety in the game prototype.

In the practical part of the thesis a prototype of a 2D shooting game was created. In the prototype the player has duels in procedurally generated levels against an opponent controlled by the Artificial Intelligence. The AI opponent adjusts itself after every defeat by examining information collected from the player. Based on the research in the theoretical part, Unity3D was chosen as the platform of the prototype.

The end result of this project was a prototype that collects information from the player and creates new AI opponents based on the information collected.

Language	Pages
Finnish	51

Keywords

dynamic difficulty adjustment, procedural content generation, Unity3D, game development

Sisällys

1 Johdanto 5
2 Dynaaminen vaikeustason säätely 6
2.1 Miten dynaamista vaikeustasoa kannattaa käyttää 6
2.1.1 Half-Life 2 ja Max Payne 7
2.1.2 Left 4 Dead -pelisarjan tekoälyohjaaja (AI Director) 7
2.2 Dynaamisen vaikeustason huonot puolet 8
2.2.1 Kuminauhatekoäly 9
2.2.2 Ice Tilt 9
2.2.3 The Elder Scrolls IV: Oblivion ja vihollisien liiallinen säätäminen 11
2.3 Yhteenveto dynaamisen vaikeustason säätelyn käytöstä 12
3 Proseduraalinen sisällön generointi 13
3.1 Proseduraalisen sisällön generoimisen käyttöesimerkkejä 13
3.1.1 Minecraft 14
3.1.2 Payday 2 14
3.1.3 Borderlands-sarja 14
3.2 Proseduraalisesti generoidun sisällön hyödyt 15
4 Testiohjelma 16
4.1 Alustan valinta 16
4.2 Testiohjelman toiminnan kuvaus 17
4.3 Tiedonkeruu 22
4.4 Tietojen tallentaminen 23
4.5 Pelaajasta kerätyn tiedon hyödyntäminen 24
4.6 Tapausnumerot ja niiden merkitykset 25
4.7 Pelialueen muokkaaminen 28
5 Algoritmikuvaukset 29
5.1 Vihollisia luova algoritmi 29
5.2 Kenttää muokkaava algoritmi 34
6 Toimivuuden testaus 38
6.1 Testin suunnittelu 38
6.2 Testin toteutus 39
6.3 Tulosten analysointi 44
7 Pohdinta 45
7.1 Idean arviointi 45
7.2 Jatkokehitysideoita 47
7.2.1 Vihollisalusten muokkaus 47
7.2.2 Kentän muokkaus 48
7.3 Yhteenveto 49
Lähteet 51

1 Johdanto

Olen jo pitkään ollut kiinnostunut tekoälystä ja sen sovellutuksista peleissä. Aloin kiinnittämään asiaan ennalta suurempaa huomiota, kun aloitin peliohjelmointiopiskelut ja ymmärsin tarkemmin, miten pelit toimivat pelatessani niitä. Huomasin miten monien pelien haaste perustuu pelaajan huijaamiseen. Nämä pelit herättivät haluni pyrkiä luomaan reiluja mutta haastavia tekoälyvastustajia peleihin, joita teen. Opinnäytetyöni aiheena on dynaaminen vaikeustason säätely ja sen soveltaminen peleissä. Kirjallisen osuuden tueksi loin testiohjelman, joka säätelee tekoälyvihollista pelaajasta kerätyn tiedon mukaan. Ohjelma sisältää myös proseduraalista generointia, jota käytetään vaihtelevuuden lisäämiseksi.

Opinnäytetyön toinen luku käsittelee dynaamista vaikeustason säätelyä. Luku sisältää perustietoa ja käyttöesimerkkejä tapauksista, joissa dynaamisen vaikeustason toteuttamisessa on joko onnistuttu tai epäonnistuttu. Kolmas luku käsittelee proseduraalista generointia. Luvussa käydään läpi käyttöesimerkkejä ja ominaisuuden tuomia hyötyjä. Kirjallisen osan neljäs luku käsittelee testiohjelmaa, jonka toiminta selitetään yksityiskohtaisesti. Viidennessä luvussa esittelen testiohjelman kaksi keskeistä algoritmikuvausta. Ensimmäinen käsittelee vihollisten säätelyä ja toinen kenttien muokkaamista. Kuudes luku käsittelee testiohjelman toimivuuden testaamista ja analysointia. Viimeinen luku sisältää pohdintaa. Luvussa arvioin testiohjelman konseptia, luodun ohjelman toimivuutta sekä jatkokehitysideoita. Lisäksi luvussa on opinnäytetyön yhteenveto.
Opinnäytetyössäni vastaan tutkimuskysymyksiin "Millainen dynaaminen vaikeustason säätely on toiminut, ja millainen ei ole toiminut?" Lisäksi vastaan tarkentavaan kysymykseen: "Miten dynaamista vaikeustason säätelyä tulisi käyttää?" Tutkin aihetta kirjallisten lähteiden ja luomani testiohjelman kautta.

2 Dynaaminen vaikeustason säätely

Dynaaminen vaikeustaso tarkoittaa sitä, että peli nostaa tai laskee vaikeustasoa riippuen pelaajan suoriutumisesta (Game Ontology 2013). Se on vaihtoehto perinteiselle staattiselle vaikeustasolle, joka on dynaamista vaikeustasoa yleisempi malli. Pelissä jossa on käytössä staattinen vaikeustaso, pelaaja valitsee pelin alussa omaa taitotasoa vastaavan vaikeustason. Dynaamisen vaikeustason idea syntyi, kun pelit monimutkaistuivat ja huomattiin, että pelaajat lopettivat pelaamisen, jos staattisen vaikeustason tuoma haaste oli joko liian suuri tai pieni (Tolentino 2008). Seuraavaksi käyn läpi, miten dynaamista vaikeustasoa kannattaa käyttää. Sen jälkeen tarkastelen dynaamisen vaikeustason huonoja puolia, ja sen luomia ongelmia. Lisäksi esittelen muutamia esimerkkitapauksia, joissa dynaamisen vaikeustason luomisessa on onnistuttu tai toisaalta ei ole onnistuttu.

2.1 Miten dynaamista vaikeustasoa kannattaa käyttää

Dynaamisen vaikeustason säätelyn käyttäminen vaatii enemmän suunnittelua ja se on monimutkaisempi toteuttaa kuin staattinen vaikeustaso. Ernest Adamsin vuonna 2008 julkaistussa kolumnissa on listattu muutamia pohdinnanarvoisia asioita dynaamisen vaikeustason tekijöille. Ensimmäiseksi kehotetaan tekemään pelistä vaikeampi vihollisia parantamalla, pelaajan heikentämisen sijaan. Pelaajaa ei saa rankaista siitä, että hän osaa pelata peliä hyvin. Vastustajien parantaminen puolestaan on hyvä ratkaisu, kunhan siitä ei seuraa epäloogisuuksia (Adams 2008.) Epäloogisuus voi olla esimerkiksi vihollisolento, joka on paljon vahvempi kuin sen pitäisi olla. On erittäin epäloogista, jos huonosti varustautunut maantierosvo on yhtä vahva kuin hyvin varustautunut ritari. Esimerkki pelaajaa rankaisevasta säätelystä on esitetty luvussa 2.2.2.

Dynaamisen vaikeustason säätelemisen ei kannata olla liian helposti pelaajan huomattavissa. Jos pelaaja ei huomaa pelin auttavan häntä, hän ei koe pelaavansa huonosti, ja pelikokemus on paljon mielekkäämpi. Vaikeustason nostamisen huomaaminen on vaikeampaa, jos se tapahtuu pienten peräkkäisten parannusten kautta. On lisäksi erittäin epäloogista, että aiemmin kohdattu vihollinen on yhtäkkiä selvästi vahvempi (Adams 2008.)

Adamsin mielestä dynaaminen vaikeustaso kannattaa lisätä ennemmin vaihtoehtoisena ominaisuutena kuin oletuksena. Vuonna 2006 julkaistussa Lego Star Wars II -pelissä pelaaja sai valita vaikeustasokseen joko staattisen tai dynaamisesti muuttuvan vaikeustason. On erittäin harvinaista, että pelissä saa valita staattisen ja dynaamisen vaikeustason väliltä. Rajallisten resurssien vuoksi pelintekijät tyytyvät yleensä vain yhden vaikeustasomallin toteuttamiseen. Dynaamisen vaikeustason käyttämisessä on tärkeää lisäksi se, että pelaaja ei tiedä mitkä tekijät vaikuttavat säätelyyn. Jos pelaaja tietää, että peli helpottuu tiettyjen kriteerien täyttyessä, hän voi käyttää vaikeustason säätelemistä hyväkseen (Adams 2008.)

2.1.1 Half-Life 2 ja Max Payne

Esimerkki hyvin piilotetusta vaikeustason säätelystä löytyy vuonna 2004 julkaistusta Half-Life 2 -pelistä. Pelistä löytyvien tarvikelaatikoiden sisällöt paranevat, jos pelihahmon osumapisteet ovat alhaiset (Tolentino 2008). Tällainen dynaaminen vaikeustason helpottaminen on niin hienovaraista, ettei pelaaja todennäköisesti huomaa sitä. Vuonna 2001 julkaistu Max Payne sisälsi myös hienovaraista dynaamista säätelyä. Sen lisäksi että pelaaja saa enemmän ammuksia ja kipupillereitä, peli auttaa tähtäämisessä, jos pelaajalla menee huonosti. Jos pelaajalla menee hyvin, peli nostaa vihollisien osumapisteiden määrää (Game Ontology 2013.)

2.1.2 Left 4 Dead -pelisarjan tekoälyohjaaja (AI Director)

Valve Corporationin luoma Left 4 Dead -pelisarja käyttää ns. tekoälyohjaajaa pelin tempon ja vaikeustason säätelemiseen. Ohjaaja ottaa huomioon pelaajien kyvyt, olinpaikan ja yleisen tilanteen. Yleisellä tilanteella tarkoitetaan tässä tapauksessa hengissä olevien ryhmäläisten määrää, terveydentilaa ja ammustilannetta. Näiden tietojen pohjalta ohjaaja säätelee normaalien vihollisjoukkojen kokoja ja niiden ilmenemispaikkoja. Lisäksi se on vastuussa ylimääräisten terveys- ja ammuspakettien, aseiden ja erikoisvihollisten lisäämisestä pelialueelle (Left 4 Dead Wiki 2014.)

Pelisarjan toisessa osassa Left 4 Dead 2 -pelissä on myös mukana pelin karttojen dynaamista säätelyä. Peliobjektien, esineiden ja sääolosuhteiden lisäksi ohjaaja voi vaikuttaa siihen, mitä reittiä pelaajien on kuljettava edetäkseen pelissä. Ohjaaja voi esimerkiksi tukkia ensisijaisen reitin tai luoda rankkasateen, joka heikentää näkyvyyttä ja äänien kantavuutta (Left 4 Dead Wiki 2014.)

2.2 Dynaamisen vaikeustason huonot puolet

Ernest Adamsin kolumnissa käsitellään myös dynaamisen vaikeustason huonoja puolia. Ensimmäinen huono puoli on pelaajien nihkeä suhtautuminen. Monet pelaajat kokevat, että peli holhoaa heitä, jos vaikeustason sääteleminen on liian ilmiselvää (Adams 2008.) Toinen dynaamisen vaikeustason huono puoli on epäloogisuuksien esiintyminen. Yksi yleinen esimerkki epäloogisesta toiminnasta, kuminauhatekoälystä, on kuvattu luvussa 2.2.1.

Kolmas huono puoli on, että dynaamista tekoälyä voi käyttää hyväksi. Pelaaja voi pelata tahallaan huonosti, jotta peli laskisi vaikeustasoa. Adamsin mukaan hyväksikäyttäminen on yksi yleisimmistä vastalauseista, mitä dynaamiselle vaikeustasolle esitetään (Adams 2008.) Neljäs huono puoli on, että dynaamista vaikeustasoa ei voi soveltaa kaikentyylisiin haasteisiin. Ominaisuuksien, kuten nopeuden ja osumapisteiden, säätäminen on helppoa, mutta esimerkiksi pulmanratkontakohtauksien säätäminen voi olla todella vaikeaa. Jotta vaikeustason säätäminen toimisi tällaisissa tilanteissa, oikeita ratkaisuja täytyy olla useampia. Lisäksi on huomioitava, että pulman luonne voi estää vaihtoehtoisten ratkaisujen lisäämisen (Adams 2008.)

Viimeinen Adamsin esittämistä dynaamisen vaikeustason ongelmista on pelin rytmin pilaantuminen. Pelaaja ei koe pelin rytmin kasvamista, koska dynaaminen vaikeustaso pitää pelin haasteen samalla tasolla koko ajan. Lisäksi dynaaminen vaikeustaso laskee hyvän kenttäsuunnittelun merkitystä, jolloin pelikokemus ei ole yhtä mieleenpainuva (Adams 2008.) Seuraavaksi käyn läpi muutaman esimerkin, joissa dynaaminen vaikeustason säätely on joko viety liian pitkälle tai siinä on muuten epäonnistuttu.

2.2.1 Kuminauhatekoäly

Moni autopeli käyttää dynaamista vaikeustason säätelyä, koska se lieventää ajovirheistä johtuvaa turhautumista. Vaikka pelaaja ajaa huonosti, hänellä on mahdollisuus saada kiinni muut kisailijat. Tällaista tekoälyä sanotaan kuminauhatekoälyksi (Game Oncology. 2013.) Kuminauhatekoälyn huonoin puoli on, että sen piilottaminen pelaajalta on erittäin vaikeaa. Kun pelaaja huomaa, että pelimenestyksen takaajana on pelin antama tasoitus, pelinautinto laskee merkittävästi. Turhauttavaa voi myös olla se, että kuminauhatekoäly estää pelaajaa ottamasta isoa etumatkaa, vaikka pelaaja ajaisi virheettömästi. Tämä puolestaan vähentää kisojen voittamisesta seuraavaa saavutuksentunnetta. Kuminauhatekoälyä on käytetty ahkerasti mm. Electronic Artsin julkaisemassa Burnout-pelisarjassa (Tolentino 2008).

2.2.2 Ice Tilt

Dynaamista vaikeustason säätelyä on myös löydetty Electronic Arts:in NHL-pelisarjasta. Pelisarjan vuonna 2008 julkaistun NHL09-pelin PC-versiosta löytyy mainintoja "Ice Tilt"-nimisestä ominaisuudesta. Ominaisuuden olemassaolosta ei ole virallista lausuntoa, mutta moni pelisarjan peleihin perehtynyt on huomannut, että peli käyttää epäreiluja keinoja ottelutilanteisiin vaikuttamiseksi (HF Boards 2013). HF Boards -sivustolla on keskusteluketju, jossa pelisarjaa pelanneet käyttäjät keskustelevat Ice Tiltin olemassaolosta. Vaikka suurin osa keskusteluun osallistuneista on sitä mieltä, että Ice Tilt on olemassa, myös vastustajia löytyy. Suurin osa on kuitenkin sitä mieltä, että ominaisuus löytyy pelistä. Lisäksi ominaisuuden olemassaoloa puoltavia videoita löytyy YouTube-videopalvelusta (VashGH 2013).

Ice Tiltin tehtävänä on muuttaa virtuaaliottelun olosuhteita siten, että siitä tulisi tasaisempi. Ice Tilt voi säätää käytännössä jokaista pelin todennäköisyyksiin tai ominaisuuksiin vaikuttavia arvoja. YouTube-käyttäjä VashGH:n videossa on esimerkkejä siitä, miten Ice Tilt toimii NHL13-pelissä. Esittelen seuraavaksi viisi videossa näytettyä Ice Tiltin esiintymistapaa.

Maalivahdin torjuntakyky

Maalivahdin torjuntakyvyn säätäminen vaikuttaa pelin kulkuun merkittävästi. Esimerkiksi jos toinen joukkue on saanut pelin alussa muutaman maalin etumatkan, Ice Tilt voi nostaa tappiolla olevan maalivahdin torjuntatodennäköisyyttä lisämaalien hankaloittamiseksi. Tämän lisäksi se voi vastaavasti tiputtaa johdossa olevan maalivahdin torjuntavalmiuksia, jolloin se saattaa päästää todella helposti muutaman maalin, ja pelitilanne tasoittuu. Maalivahdin säätely on yksi helpoimmin havaittavista tapauksista, joissa pelaaja voi huomata, että peli päättää tasoittaa tilannetta. Se voi olla erittäin turhauttavaa jos pelaaja selvästi hallitsee peliä, mutta vastustaja pysyy tulostaululla taistelussa mukana pelin toteuttaman epäreilun kikkailun vuoksi.

Pelaajien jäähyherkkyys ja karvauksien onnistuminen

Jäähyherkkyys on myös yksi helposti havaittava Ice Tiltin säätelemä ominaisuus. Koska videopelin tuomarit eivät oikeasti "näe", millainen taklaus- tai karvaustilanne oli, kaikki toimii todennäköisyyksiä laskemalla. Pelaaja voi taistella kiekosta joko taklaamalla tai käyttämällä mailahäirintää. Kumpikin näistä teoista voi aiheuttaa jäähyn, eikä pelaaja voi kunnolla vaikuttaa karvausyritysten säännönmukaisuuteen. Kun pelaaja suorittaa jonkinlaisen karvauksen, peli laskee, miten karvausyritys vaikuttaa tilanteen osapuoliin. Lisäksi peli laskee, tuomitaanko tilanteesta jäähyä. On siis mahdollista, että pelaaja kaataa vastustajan yrittäessään mailalla riistää kiekkoa, eikä tästä vihelletä jäähyä. Vastaavanlaisesti täysin puhtaasta taklauksesta voidaan viheltää rangaistus, mikäli toinen pelaaja on ollut pelissä alakynnessä. Ice Tilt voi siis vaikuttaa siihen, kuinka hyvin karvausyritykset onnistuvat ja kuinka helposti karvausyrityksistä voi tulla jäähy.

Irtokiekon saaminen

Ice Tilt voi myös vaikuttaa siihen, kumpi joukkue saa haltuunsa irtokiekkoja. Irtokiekkojen saaminen on erittäin tärkeää esimerkiksi maalinedustatilanteissa, joissa irtokiekon saaminen voi ratkaista pelin. Se osapuoli jota Ice Tilt tukee saa todennäköisemmin kiekon haltuunsa, kun kiekon lähettyvillä on useampi mahdollinen ottaja. Lisäksi useissa pelivideoissa näkyy, kuinka mailahäirinnän kautta irronnut kiekko hakeutuu epäluonnollisen näköisesti suoraan kiekon menettäneen osapuolen pelaajan lapaan. Ice Tiltin

voidaan tällaisissa tapauksissa sanoa haluavan, että kiekko pysyy tietyn osapuolen hallussa.

Syöttöjen kaappaamisen todennäköisyys

Syöttöjen kaappaamisen todennäköisyys on irtokiekkojen saamisen kanssa yksi vaikeammin havaittavista Ice Tiltin vaikutuksista. Kummankin havaitseminen vaatii useita pelitunteja tai tarkkasilmäisen pelaajan. Peliä tarkasti seuraamalla voi huomata, miten syötöt joskus menevät puolustajien läpi ilman mitään syytä. Monelta videolta näkyy, miten puolustaja "näkee" syötön lähtevän, mutta syöttö menee puolustajan läpi. Jos Ice Tilt toimii puolustavaa joukkuetta vastaan, puolustajat eivät saa mitään kiinni ja kiekon vastaanottamisyritykset ovat erittäin ala-arvoisia. Lisäksi kiekko monesti lipuu joko puolustajan luistinta tai mailaa hipoen takana odottavan vastustajan lapaan. Vastaisuudessa omat samanlaiset syöttöyritykset jäävät armotta vastustajan puolustajan lapaan, tehden monet syöttömahdollisuudet mahdottomiksi jo ennen syötön lähtemistä.

2.2.3 The Elder Scrolls IV: Oblivion ja vihollisien liiallinen säätäminen

Bethesda Game Studios päätti käyttää dynaamista vaikeustasoa vuonna 2006 julkaistussa The Elder Scrolls IV: Oblivion - pelissä. Sen lisäksi että dynaamista vaikeustasoa käytettiin pelin vihollisien tason parantamiseen, pelaajan taso vaikutti myös siihen, mitä vihollisia pelimaailmasta löytyy. Pelaajan tason kasvaessa pelin heikommat viholliset korvattiin uusilla vahvemmilla vihollisilla. Tästä seurasi ongelmia tilanteissa joissa pelaaja tarvitsi edetäkseen esimerkiksi suden taljoja, mutta pelaajan tason vuoksi susia ei pelimaailmasta löydy. Ongelmia ilmeni myös, kun pelaajan piti auttaa tietokoneen ohjastamia hahmoja. Pelaajan korkean tason vuoksi kohdatut viholliset olivat liian suuri haaste tietokoneen ohjastamille sotureille. Pelin tehtäväsuunnittelu ja dynaaminen vaikeustaso eivät siis toimineet yhdessä. Lisäksi järjestelmän vuoksi pelin läpäiseminen oli todella helppoa, jos pelaaja päätti olla nostamatta omaa tasoansa (Tolentino 2008.)

The Elder Scrolls IV: Oblivionin dynaaminen vaikeustason säätely on liian ilmiselvää ja tehokasta. Tästä syntyi epäloogisuuksia, jotka ovat yksi Adamsin mainitsemista dynaamisen vaikeustason huonoista puolista. TES IV: Oblivionin sisältämä säätely olisi voi-

nut olla paremmin onnistunutta, jos vihollisien suoraviivaisen parantamisen sijaan olisi esimerkiksi säädelty vihollisjoukkojen kokoa.

2.3 Yhteenveto dynaamisen vaikeustason säätelyn käytöstä

Taulukko 1. Dynaamisen vaikeustason säätelyn ominaisuuksia.

Tee näin
Tee ominaisuudesta vapaaehtoinen.
Piilota ominaisuuden olemassaolo.
Älä rankaise pelaajaa.
Älä mahdollista epäloogisuuksia.
Älä luo liian vahvaa kuminauhaefektiä.

Taulukossa 1 on koostettuna muutamia ohjeita, joita kannattaa pohtia dynaamista vaikeustasoa suunniteltaessa. Ensimmäinen ohje on ominaisuuden valitsemisen vapaaehtoisuus. Jos dynaaminen vaikeustaso on vapaasti valittavissa, ne pelaajat jotka pitävät enemmän perinteisestä staattisesta vaikeustasosta voivat suhtautua peliin paljon myönteisemmin. On myös mahdollista, että varsinkin huonosti toteutettu dynaaminen vaikeustaso voi vaikuttaa ostopäätökseen. Näin voi käydä ainakin autopelien kohdalla. Toinen ohje on dynaamisen vaikeustason piilottaminen. Varsinkin niissä tapauksissa joissa dynaamisen vaikeustaso on ainoa vaihtoehto, on tärkeää panostaa ominaisuuden näkymättömyyteen. Jos ominaisuus on liian ilmiselvä, pelaajasta voi tuntua siltä, että peli pelaa itse itseään, kun pelaajan taidoilla ei ole väliä.

Vältettävistä ominaisuuksista ensimmäinen on pelaajan rankaiseminen. Pelin vaikeustasoa ei saa nostaa niin paljoa, että menestymismahdollisuudet ovat liian pieniä. Esimerkiksi aiemmin esitelty Ice Tilt saattaa aiheuttaa pelaajalle häviöitä, jos pelaaja on pelannut liian hyvin. Toinen vältettävä ominaisuus on epäloogisuudet. Jos peliä vaikeutetaan ainoastaan vihollisyksilöitä parantamalla, on mahdollista, että jossain välissä viholliset ovat epäloogisen vahvoja. Näin käy esimerkiksi aiemmin mainitussa The Elder Scrolls IV: Oblivion - pelissä. Epäloogisuuksiin kuuluu myös kuminauhaefekti. Vaikka dynaamiseen vaikeustasoon kuuluu myös vaikeustason laskeminen, on erittäin tärkeää muistaa, että liian ilmiselvä pelin helpottaminen vaikuttaa negatiivisesti pelinautintoon. Jos

dynaamisen vaikeustason piilottaminen ei onnistu kokonaisuudessaan, niin ainakin peliä helpottavien toimien tulisi pysyä piilossa. Vaikeustason nouseminen pelin aikana on normaalia, mutta sen laskemista voidaan pitää pelaajan taitoja loukkaavana.

3 Proseduraalinen sisällön generointi

Proseduraalisella sisällön generoimisella tarkoitetaan, että peliin luodaan sisältöä käyttäen tietokonealgoritmia, johon käyttäjä voi vaikuttaa rajatusti tai epäsuorasti (Nelson, Togelius & Shaker 2013). Tällaista algoritmia käyttävät pelit voivat luoda uutta sisältöä pelin ollessa käynnissä jokaisella pelikerralla. Tästä johtuen pelin uudelleenpeluuarvo nousee merkittävästi, koska jokainen pelikerta on erilainen. Nelson, Togelius ja Shaker (2013) sanovat kirjassaan, että termin "proseduraalisen sisällön generointi" avainsana on sisältö. He määrittelevät sisällöksi suurimman osan pelistä, esimerkiksi kentät, säännöt, äänet tarinat hahmot jne. Sisällöksi ei lasketa pelin tekoälyä.

3.1 Proseduraalisen sisällön generoimisen käyttöesimerkkejä

Proseduraalista sisällön generoimista voidaan hyödyntää kahdessa vaiheessa pelinkehitystä. Ensinnäkin sitä voi hyödyntää osana lopputuotetta, jolloin algoritmi luo uutta sisältöä valmiiseen peliin jatkuvasti. Toiseksi sitä voidaan hyödyntää pelituotannon aikana korvaamaan työntekijän työpanosta, luomaan esimerkiksi pelikenttiä, jotka lisätään normaaliin testauskiertoon (Coleman 2011). Proseduraalisen generoimisen algoritmi ei välttämättä luo sisältöä nopeasti, joka tarkoittaa käytännössä sitä, ettei sitä voi käyttää samanaikaisesti pelin ajamisen aikana. Tällöin algoritmia voidaan esimerkiksi käyttää jo pelituotannon aikana ja tallentaa toimiviksi katsotut ratkaisut joko pelilevylle tai siihen varatulle palvelimelle. Tällöin saadaan koko hyöty proseduraalisesti generoidusta sisällöstä ilman suuria aika- ja suorituskykyvaatimuksia pelitilanteen aikana.

3.1.1 Minecraft

Proseduraalisesti generoitua sisältöä on käytetty tähän mennessä lähinnä pelialueiden ja kenttien luomiseen tai valmiiksi luotujen kenttien sisältämien olosuhteiden muuttamiseen. Proseduraalisesti generoituja pelimaailmoja käyttävistä peleistä yksi hyvä esimerkki on Mojang-studion vuonna 2011 julkaisema Minecraft, jossa peli luo massiivisen maailman, missä pelaaja voi seikkailla mielensä mukaisesti. Vaikka proseduraalisen generoinnin algoritmit sisältävät tietyt säännöt, Minecraftin käyttämä algoritmi tuntuu pelaajan näkökulmasta siltä, ettei mikään pelin sisällä oleva yksittäinen osa-alue rajoita pelimaailman luontia (Mojang 2011.)

3.1.2 Payday 2

Valmiiden pelialueiden sisältämien olosuhteiden muokkaamisesta on useita esimerkkejä, kuten Overkill Software -studion vuonna 2013 tekemä Payday 2. Payday 2:ssa pelaajien tehtävänä on suorittaa mm. pankkiryöstöjä ja peli vaihtaa pankkiholvien, turvakameroiden ja vartioiden paikkoja ja käyttäytymisiä niin, että pelaaja ei voi koskaan tietää mitä odottaa. Vaikka proseduraalisen generoinnin algoritmi muokkaa pelikokemusta, on huomioitava, että tällaisissa tapauksissa – toisin kuin esimerkiksi Minecraft:ssa – on mahdollista nähdä kaikki mahdolliset lopputulokset. Algoritmi ei siis luo täysin tyhjästä sisältöä vaan valitsee valmiiksi luoduista vaihtoehdoista. Tällaisissa tapauksissa algoritmi voi käyttää apuna satunnaislukugeneraattoria tai sillä voi olla omia tarkentavia sääntöjä joiden perusteella sisältö valitaan ja rajoitetaan (Overkill Software 2013.

3.1.3 Borderlands-sarja

Gearbox Softwaren vuonna 2012 julkaistu peli Borderlands 2 käyttää proseduraalista generoimista pelin ase- ja varustevalikoiman luomiseen. Ominaisuuksien lisäksi niistä muuttuu niin aseen kuin sen ammusten ulkomuoto ja väritys. Pelisarjan ensimmäinen osa Borderlands(2010) pitää hallussaan Guinnessin maailmanennätystä suurimmasta asemäärästä videopelissä. Pelin tekijöiden mukaan jatko-osassa on enemmän aseita kuin ensimmäisessä, mutta tarkka lukumäärä ei ole tiedossa (Yin-Poole 2012).

Borderlands-peleissä proseduraalisesti generoitavien objektien luontia rajoittaa ohjelmoitujen sääntöjen lisäksi pelaajan taso pelissä. Pelaaja siis saa koko ajan omaan tasoonsa sopivia varusteita. Taulukossa 2 on esitetty saman aseen kaksi erilaista ilmenemistapaa.

Taulukko 2. Borderlands 2, Asevertailu (Borderlands 2 wiki 2013).

Nimi:	Consummate Hellfire	Apt Hellfire
Taso:	50	50
Vahinko:	4363	4092
Tarkkuus:	93,1	92,3
Tulinopeus:	8,0	8,0
Latausnopeus:	2,7	2,8
Lipaskoko:	29	37
Tulivahinko/sekunti:	4467,5	4467,5
Sytyttämismahdollisuus:	18,8 %	18,8 %

3.2 Proseduraalisesti generoidun sisällön hyödyt

Proseduraalisesti sisältöä generoivia algoritmeja käytetään peleissä lisäämään uudelleenpeluuarvoa ja pidentämään pelin ikää. Koska algoritmi luo aina erilaisia yhdistelmiä, (tai ainakin mahdollisia yhdistelmiä on tuhansia) jokainen pelikerta on erilainen, ja peleistä riittää viihdettä pidemmäksi aikaa. Joissakin tapauksissa on mahdollista, että pelaaja kokee kaikki mahdolliset yhdistelmät, mutta siinä on kuitenkin todennäköisesti mennyt pidempään, kuin olisi mennyt yhden kaikille saman tehdyn pelin läpäisemisessä.

Nykypäivänä pelijulkaisijoiden yksi suurimmista haasteista on saada pelin ostaneet kuluttajat säilyttämään pelin hyllyssään sen myymisen sijaan läpäisyn jälkeen. Tämä johtuu kahdesta asiasta: käytettyjen pelien myymisestä ei tule tuottoja julkaisijoille ja peleihin julkaisun jälkeen tehtävät lisäsisältöpaketit voidaan myydä isommalla voittomarginaalilla. Jos peli käyttää proseduraalisesti sisältöä generoivaa algoritmia pelin iän pidentämiseen, on todennäköisempää, ettei peliä viedä heti vaihdettavaksi. Mikäli peli turvautuu todella vahvasti proseduraalisen sisällön generaation algoritmiin, siihen luo-

tava lisäsisältö voi tuoda lisää uusia mahdollisia yhdistelmiä tai jopa tehdä emopelin sisällöstä jonkin verran tuoreempaa.

4 Testiohjelma

Testiohjelman lajityyppi on klassinen ylhäältäpäin kuvattu kahden analogisauvan ammuntapeli. Testiohjelmassa käyttäjä ohjaa pyöreää hahmoa suorakaiteen muotoisen pelialueen sisällä, taistellen jokaisen kohtaamisen jälkeen vahvemmiksi kasvavia vihollisaluksia vastaan. Valitsin kyseisen lajityypin, koska sen perusmekaniikat ovat helppoja toteuttaa ja voin keskittyä tutkittavan aiheen toteutukseen.

4.1 Alustan valinta

Opinnäytetyöhön kuuluva testiohjelma toteutetaan Unity3D-pelimoottoriin. Valitsin Unityn testiohjelman alustaksi sen helppokäyttöisyyden, aikaisemman kokemuksen ja vapaasti käytettävien peliassettien vuoksi. Unity3D tukee kolmea eri ohjelmointikieltä: vahvasti JavaScript-kielen tapaista UnityScriptiä, C#:ia ja Boo-ohjelmointikieltä (Unity Technologies 2013.)

Ohjelmoin testiohjelman C#:lla, koska minulla on siitä aiempaa kokemusta, ja olen muutenkin aina käyttänyt sitä Unity3D:n kanssa. Alustavalintoja koskien päätin pysyä tuntemieni ratkaisujen parissa, koska silloin voin täysin keskittyä itse testiohjelman tekemiseen enkä uuden alustan opettelemiseen.

Laajassa käytössä olevan Unreal Engine 3:n ilmaisversio Unreal Development Kit olisi myös voinut olla yksi vaihtoehto testiohjelman alustaksi, mutta uuden tuntemattoman moottorin käytön opettelu olisi lisännyt työmäärää liikaa. Lisäksi Unreal Development Kit käyttää omaa uniikkia skriptauskieltä UnrealScriptiä, joka on samantyylinen C++- ja Java - kielien kanssa (Epic Games 2013.) Vaikka minulta on jonkin verran kokemusta C++- ja Java kielistä, uuteen skriptauskieleen perehtyminen ei ole tällä hetkellä kannattavaa.

Unity3D oli selvä valinta pelimoottorien kannalta, mutta testiohjelma olisi voitu luoda myös käyttäen Microsoftin DirectX-rajapintaa tai sen päälle rakennettua XNA-kirjastoa. XNA-kirjaston käyttö ei enää ole suositeltava tai hyvä ratkaisu, sillä Microsoft on päättänyt lopettaa sen tukemisen kokonaan, eli sen voidaan katsoa poistuneen käytöstä (Rose 2013). DirectX puolestaan olisi muuten hyvä ratkaisu, mutta se lisäisi työmäärää merkittävästi, koska sitä käyttäessä joutuisi luomaan käytännössä kaiken tyhjästä.

4.2 Testiohjelman toiminnan kuvaus

Testiohjelmassa käyttäjä saa itse säätää muutamaa pelattavuuteen vaikuttavaa ominaisuutta: aluksen ase, aseeseen vaikuttava vahinkobonus, aluksen nopeusbonus ja aluksen kestävyysbonus. Antamalla käyttäjälle valinnanvaraa varmistetaan, että jokainen pelikerta on erilainen heti sen alkaessa.

Asevalikoima (taulukko 3) koostuu kolmesta pohjimmiltaan samanlaisesta peliobjektista, mutta niiden ominaisuudet on tasapainotettu eri tavalla. Ensimmäinen ase on normaali konekivääri, jolla voi ampua viisi ammusta nopeasti ja tarkasti. Aseen heikkouksia ovat pieni vahinkopotentiaali, ja viidennen laukauksen jälkeen seuraava pitkä lataustauko. Toinen ase on haulikko, joka ampuu viisi ammusta kerrallaan. Haulikon heikkouksia ovat ammusten satunnainen lentorata ja pitkä lataustauko. Viimeinen ase on raidetykki, joka ampuu erittäin nopeita ja suurta vahinkoa aiheuttavia ammuksia. Sen heikkous on pitkä latausaika.

Taulukko 3. Aseiden oletus-ominaisuudet.

Aseen nimi	Vahinko	Ammuksen nopeus	Latausväli	Erikoisparannuskohde
Haulikko	10,0 f	50,0 f	1,5 f	Haulien määrä
Konekivääri	10,0 f	50,0 f	0,08f	Lipaskoko
Raidetykki	25,0 f	65,0 f	2,5 f	Latausaika

Taulukossa 3 esitetyistä ominaisuuksista ensimmäinen on aseen vahinko. Vahinko vaikuttaa siihen kuinka monta osumapistettä osuman kohteelta vähennetään. Ammuksen nopeus viitaa siihen kuinka nopeaa ammus lentää. Latausväli viittaa taukoon, joka on

jokaisen ammuksen välissä. Esimerkiksi konekiväärin latausväli on pienin, koska se ampuu useita ammuksia peräkkäin.

Aseen lisäksi pelaaja saa laittaa mieleiseensä järjestykseen kolme ominaisuusbonusta: vahingon, kestävyyden ja nopeuden. Vahinkobonus nostaa pelaajan aseen tehoa, kestävyysbonus nostaa aluksen osumapisteiden määrää, ja nopeusbonus nostaa aluksen liikkumisnopeutta. Bonusten prosenttiarvot on esitetty taulukossa 4.

Taulukko 4. Bonusten prosenttiarvot.

Bonus	Valinta 1	Valinta 2	Valinta 3
Vahinko	30 %	20 %	10 %
Kestävyys	60 %	40 %	20 %
Nopeus	75 %	50 %	25 %

Taulukossa 4 esitetyt bonukset lisätään pelaajan aluksen ominaisuuksiin, riippuen siitä missä järjestyksessä hän valitsee ne. Vahinkobonus lisätään suoraan käytössä olevan aseen vahinkoarvoon, kestävyysbonus nostaa pelaajan osumapisteiden määrää ja nopeus korottaa pelaajan aluksen liikkumisvauhtia. Esimerkkejä bonusten vaikutuksista ominaisuuksien lähtöarvoihin on esitetty taulukossa 5.

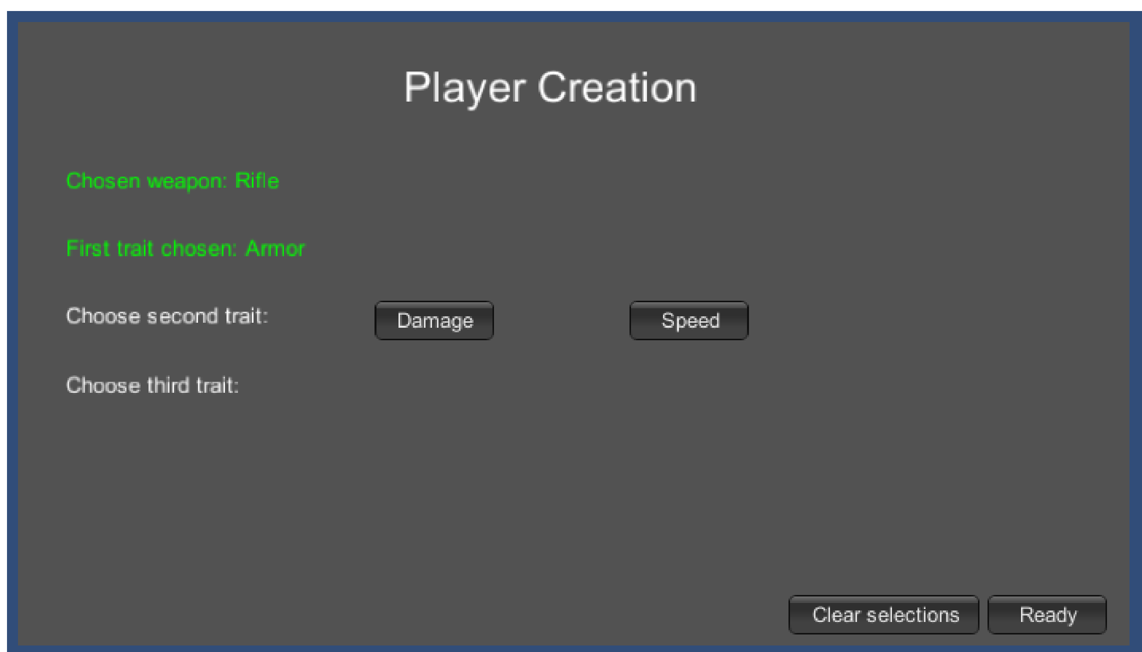
Taulukko 5. Bonusten vaikutuksien esimerkkitapauksia.

Ase	Bonus 1: Vahinko	Bonus 2: Kestävyys	Bonus 3: Nopeus
Haulikko	10 x 5 => 13 x 5	100 => 140	6 => 7,5
Ase	Bonus 1: Kestävyys	Bonus 2: Nopeus	Bonus 3: Vahinko
Konekivääri	100 => 160	6 => 9	10 => 11
Ase	Bonus 1: Nopeus	Bonus 2: Vahinko	Bonus 3: Kestävyys
Raidetykki	6 => 10,5	25 => 30	100 => 120

Testiohjelman pelaaminen alkaa valikkonäkymästä, jonka kautta käyttäjä voi jatkaa itse pelinäkymään, lukea tietoja testiohjelmasta ja sulkea ohjelman (kuva 1).

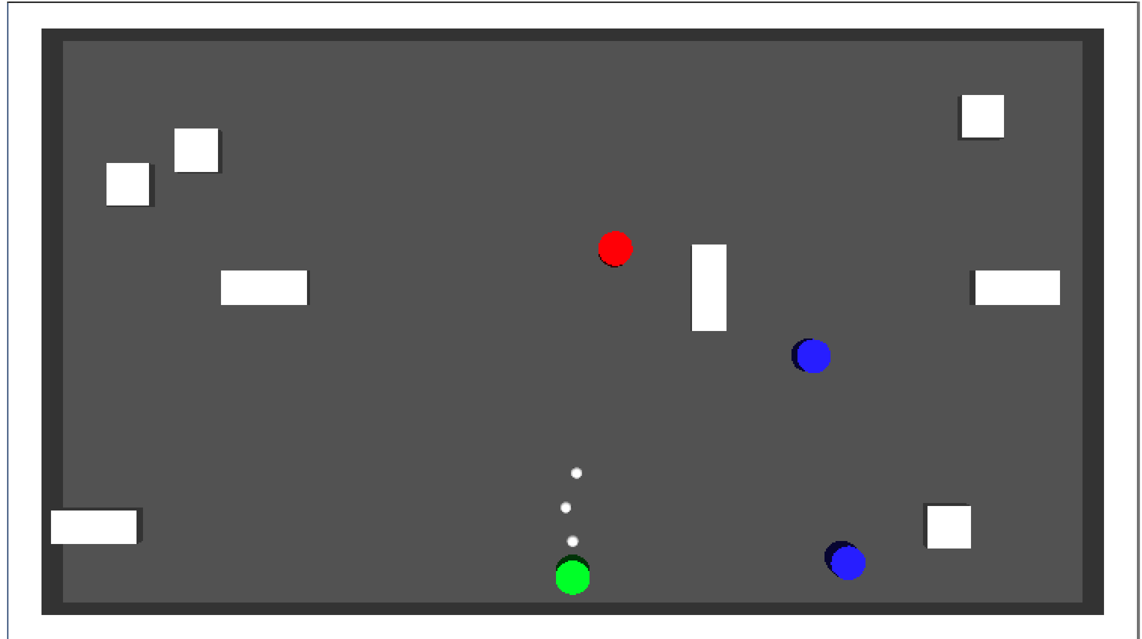
Kuva 1. Päävalikko.

Kun käyttäjä aloittaa varsinaisen pelisession, hänet ohjataan valintaruutuun, jossa hän tekee pelihahmoa koskevat valinnat (kuva 2).



Kuva 2. Pelaajanluonti.

Kun pelaaja on tehnyt tarvittavat valinnat, siirrytään ensimmäistä kertaa itse pelinäkymään, jossa käyttäjä kohtaa ensimmäisen vihollisaluksen (kuva 3).

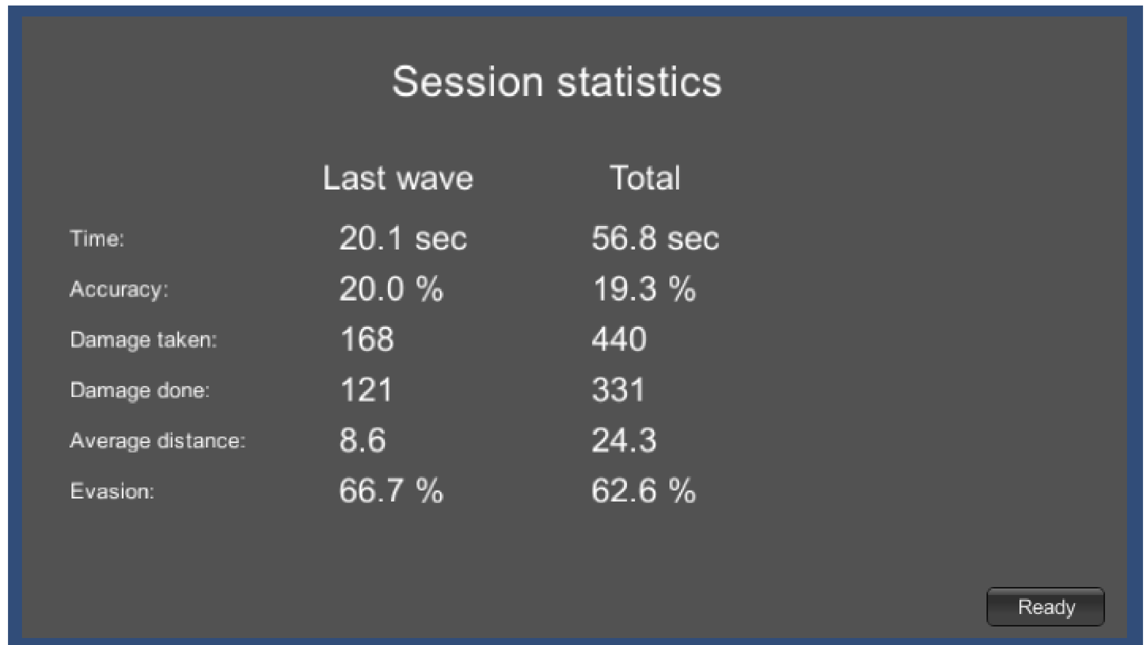


Kuva 3. Taistelunäkymä.

Vaikka jokainen kierros pelataan samalla kentällä, jokainen taistelu on siitä huolimatta erilainen. Testiohjelma käyttää proseduraalisen generaation algoritmia muuttamaan kentän olosuhteita. Ennen jokaisen taistelukohtauksen alkua algoritmi asettelee kenttään esteitä ja tykkitorneja. Esteet tarjoavat lisää suojaa tulitukselta esimerkiksi aseen lataamisen ajaksi. Tykkitornit puolestaan ovat aluksi toimettomassa tilassa, mutta ne vastaavat tulitukseen, jos pelaaja tai vihollinen osuu niihin. Mikäli molemmat osapuolet ovat hyökänneet tykkitornin kimppuun, tykkitorni ottaa kohteekseen lähempänä olevan osapuolen.

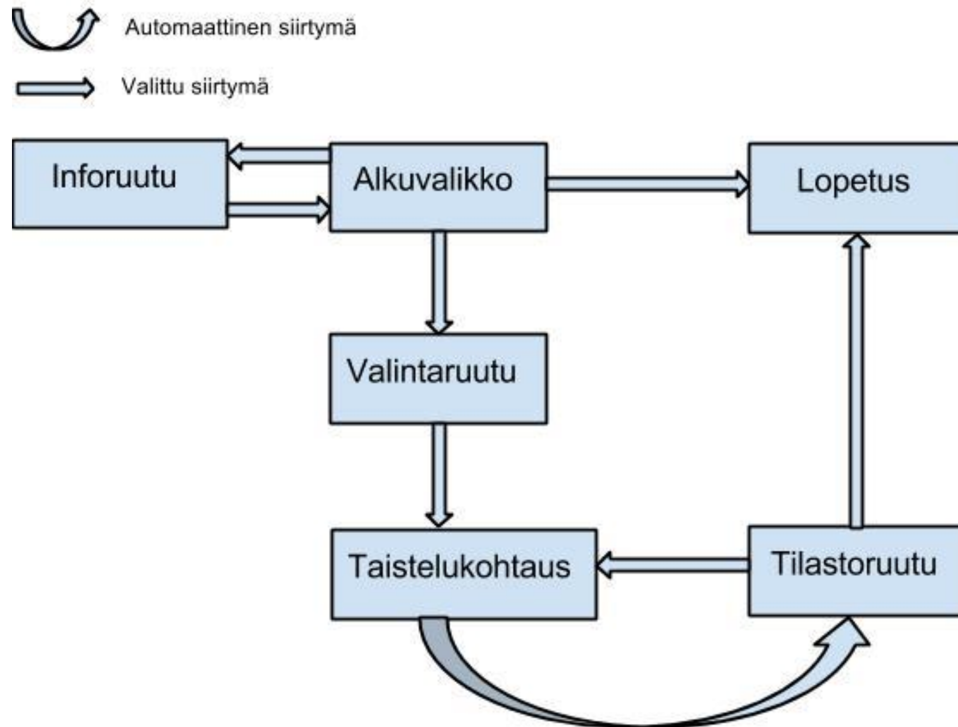
Pelin edetessä algoritmi nostaa kenttään lisättävien ekstra-objektien määrää. Seiniä lisätään ensimmäisellä kierroksella kolme, ja määrä kasvaa jokaisella kierroksella yhdellä. Tykkitorneja lisätään joka toinen kierros alkaen kierrokselta kaksi. Viimeisellä kierroksella kentälle lisätään siis 13 seinäobjektia ja viisi tykkitornia. Kenttää muokkaavaa algoritmia käsitellään tarkemmin luvussa 5.2.

Kun pelaaja selvittää ensimmäisen taistelukohtauksen, siirrytään taukonäkymään, jossa pelaaja näkee valikoituja tietoja edellisestä taistelusta. Taukonäkymä on teknisesti samanlainen kuin valintaruutu, mutta siinä näytetään enemmän tietoja, eikä pelihahmoa voi enää muokata (kuva 4). Kun pelaaja painaa oikean alakulman "Ready"-painiketta, seuraava toimintakohtaus alkaa.



Kuva 4. Tilastonäkymä.

Painikkeen painamisen jälkeen testiohjelma tulkitsee edellisestä kierroksesta kerättyjä tietoja. Tiedoista tehdyt johtopäätökset välitetään parametreinä proseduraalisen generoinnin algoritmille, joka luo uuden vihollisen pelaajan vastukseksi. Kameran siirtyessä takaisin taistelukentälle, pelaajan vastassa on uusi paranneltu vihollisalus, ja uudelleen muokattu kenttä. Tästä muodostuu testiohjelman keskeinen kierto, joka toistetaan kunnes pelaaja on selvittänyt kymmenen taistelua. Testiohjelman täysi kulkukaavio on esitetty kuvassa 5.



Kuva 5. Testiohjelman kulkukaavio.

Pelaaja kohtaa ensin kolme kertaa kiväärillä varustettuja vihollisia. Kolmen kierroksen jälkeen ase vaihtuu haulikkoon, ja vihollisobjektien ominaisuudet palautetaan alkuperäiselle tasolle. Haulikon jälkeen prosessi toistetaan viimeisen kerran raidetykin kanssa. Jokaisella aseella varustettua vihollista parannetaan kolme kertaa, mutta pelaaja kohtaa viimeiset versiot vasta viimeisellä kierroksella. Kymmenes kierros on ainoa kierros, jossa pelaaja kohtaa useamman vihollisen ilman oman aluksen korjautumista. Kierroksilla 1–9 pelaaja kohtaa yhden vihollisaluksen kerrallaan, ja jokaisen kierroksen välissä pelaajan aluksen osumapisteitä palautetaan puolet maksimitasosta vastaava määrä. Viimeisellä kierroksella pelaaja kohtaa 3 alusta peräkkäin ilman taukoa tai osumapisteiden palautusta.

4.3 Tiedonkeruu

Jotta testiohjelma voi parantaa vihollisobjektien ominaisuuksia, pelitapahtuman aikana pitää kerätä seuraavia tietoja: kulunut aika sekä pelaajan ja vihollisen osumatarkkuus (taulukko 6).

Taulukko 6. Kerättävät tiedot.

Kerättävä Tieto	Pelaaja	Vihollinen	Jaettu
Kulunut aika			X
Osumatarkkuus	X	X	

Käytetty aika on ensimmäinen tieto, mitä tutkitaan ennen kuin tehdään päätöksiä. Lisäksi se on yksi pelaajalle esiteltävistä tiedoista taukoruudussa. Käytetyn ajan seuraaminen voisi olla yksi tapa selvittää, onko tekoälyn tarjoama vastustus kasvanut, mutta testiohjelman nykyisestä versiosta löytyvät tykkitornit voivat vääristää tuloksia. Lisäksi testiohjelman ulkopuolinen tekijä, kuten pelaajan keskittymisen herpaantuminen tai mielenkiinnon tippuminen, voivat vaikuttaa tuloksiin. Kuluneen ajan seuraaminen on erittäin helppoa. Se saadaan selville luomalla yksinkertainen funktio, joka laskee aikaa taistelukohtauksen alusta viimeisen vihollisaluksen tuhoutumiseen asti.

Osumatarkkuudet ovat käytetyn ajan jälkeen seuraavaksi tarkisteltavat tiedot. Vihollisen osumatarkkuus vaikuttaa siihen, miten aluksen asetta parannetaan seuraavalle kierrokselle. Pelaajan osumatarkkuus eli vihollisaluksen väistöprosentti puolestaan päättää siitä, miten vihollisaluksen puolustusta parannetaan. Lisäksi molemmat tiedot vaikuttavat siihen, miten läheltä vihollinen hakee seuraavaa määränpäätä. Osumatarkkuuksien laskeminen hoituu erittäin yksinkertaisesti. Tiedonkeruusta vastaava olio laskee jokaisen osuman ja selvittää niiden suhteen ammuttujen laukausten kokonaismäärään.

4.4 Tietojen tallentaminen

Opinnäytetyön testiohjelma tallentaa pelaajasta kerätyn tiedon ohjelmakoodin kautta luotaviin tyhjiin peliobjekteihin. Tieto pitää tallentaa, koska sitä tarvitaan muutosten vaikutusten arviointiin. Koska vain kahden edellisen kierroksen tietoja vertaillaan, osa tiedoista muuttuu tarpeettomiksi. Lisäksi pelaajalle näytettäväksi kerättävä kokonaistilasto on testiohjelman toiminnan kannalta tarpeeton.

Tietojen tallentaminen testiohjelman ulkopuolelle olisi erittäin hyvä ominaisuus, mutta en katsonut sitä tarpeelliseksi tässä vaiheessa. Opinnäytetyön luvussa 6 (Toimivuuden

testaus) esitetyt ja analysoidut tiedot on otettu talteen vähemmän käytännöllisesti kopioimalla kuvankaappauksista.

4.5 Pelaajasta kerätyn tiedon hyödyntäminen

Taistelukohtauksen jälkeen testiohjelma aloittaa kerätyn tiedon käsittelyn ja tutkimisen. Ensimmäiseksi ohjelma muuntaa kerätyt tiedot sellaiseen muotoon, että niitä voidaan vertailla. Kun tiedot on muunnettu ja keskiarvot laskettu, niitä verrataan ennalta määriteltyihin minimitavoitteisiin. Minimitavoitteissa onnistuminen määrittää ns. tapausnumeron, jota ohjelma käyttää vihollisen säätämisessä. Säädettävät ominaisuudet on jaettu kahteen alakategoriaan: puolustus ja hyökkäys. Puolustukseen kuuluvat vihollisaluksen panssari, liikkumisnopeus ja etäisyys pelaajasta. Hyökkäykseen kuuluvat aseen tulivoima, ammusten nopeus ja jokaisen aseen oma erityisominaisuus. Parannettavat ominaisuudet on listattu taulukossa 7.

Taulukko 7. Parannettavat ominaisuudet.

Kohde	**Puolustus**	**Ase**
Ominaisuus 1	Panssari	Tulivoima
Ominaisuus 2	Nopeus	Ammusnopeus
Ominaisuus 3	Etäisyys	Aseesta riippuvainen*

*Katso taulukko 3.

Ajan minimitavoite on selvitä vähintään 20 sekuntia pelaajaa vastaan. Jos aikatavoite ei täyty, säätövaiheessa keskitytään ensisijaisesti puolustuksien kehittämiseen. Mikäli tavoite täyttyy, säätelyssä voidaan keskittyä tuhovoiman parantamiseen. Aikatavoitteen lisäksi tutkitaan vihollisaluksen osumatarkkuutta ja väistöprosenttia. Vihollisobjekti pyrkii siihen, että osumatarkkuus on vähintään 33 %, ja väistöprosentti eli pelaajan osumaprosentti korkeintaan 33 %. Osumatarkkuuksien suhteen on löydettävä sopiva tasapaino, joka saavutetaan joko säätämällä vihollisobjektin etäisyyttä pelaajasta tai aseen ammusnopeutta nostamalla. Lisäksi haulikon ja konekiväärin erityisominaisuuksien parantamisella voi olla suora vaikutus osumatarkkuuteen.

4.6 Tapausnumerot ja niiden merkitykset

Taulukko 8. Tapausnumeroiden selvitykset ja parannettavat ominaisuudet.

	Tavoitteiden onnistuminen			**Parannettavat ominaisuudet**	
Tapaus	**Aika**	**Väistäminen**	**Osuminen**	**Puolustus**	**Hyökkäys**
111	Ei	Ei	Ei	Nopeus +2 Kestävyys +1 Etäisyys +1	Ammusten nopeus +1 Erikoisominaisuus +1
121	Ei	Kyllä	Ei	Nopeus +1 Kestävyys +1	Etäisyys -2 Ammusten nopeus +1 Erikoisominaisuus +1
112	Ei	Ei	Kyllä	Nopeus +2 Kestävyys +1 Etäisyys +1	Vahinko +1 Erikoisominaisuus +1
122	Ei	Kyllä	Kyllä	Kestävyys +3 Nopeus +1	Vahinko +1 Erikoisominaisuus +1
211	Kyllä	Ei	Ei	Nopeus +1 Kestävyys +1	Ammusten nopeus +2 Erikoisominaisuus +1 Vahinko +1
221	Kyllä	Kyllä	Ei	Kestävyys +1	Ammusten nopeus +2 Erikoisominaisuus +1 Vahinko +1 Etäisyys -1
212	Kyllä	Ei	Kyllä	Etäisyys +1 Nopeus +1	Vahinko +2 Ammusten nopeus +1 Erikoisominaisuus +1
222	Kyllä	Kyllä	Kyllä	Nopeus +1 Kestävyys +1	Vahinko +2 Ammusten nopeus +1 Erikoisominaisuus +1

Taulukossa 8 listatut tapausnumerot muodostuvat onnistuneista tavoitteista siten, että jokainen onnistunut tavoite merkitään numerolla 2 ja epäonnistunut tavoite numerolla 1. Numerot laitetaan tärkeysjärjestykseen, minkä jälkeen niistä muodostuu kolminumeroinen luku. Tavoitteiden tärkeysjärjestys on aika, väistö ja osuma, eli jos vihollinen onnistuu aika- ja osumatavoitteissaan tapausnumeroksi muodostuu 212. Jos vihollinen epäonnistuu kaikissa tavoitteissa, numeroksi muodostuu 111.

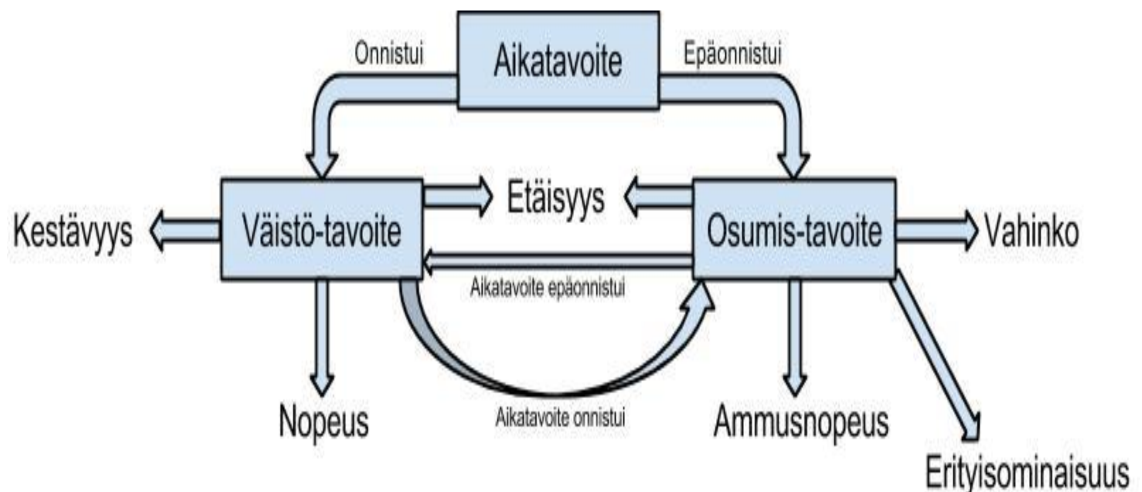
Tapausnumerot voidaan jakaa kahteen eri alakategoriaan: puolustukseen ja hyökkäykseen keskittyviin tapausnumeroihin. Se, kumpaan kategoriaan mikäkin tapausnumero kuuluu, selviää sen ensimmäisestä luvusta. Jos ensimmäinen luku on 1, kyseessä on puolustukseen keskittyvä tapausnumero. Toisin sanoen aikatavoite päättää siitä, mihin kategoriaan tapausnumero kuuluu, ja siitä, kuinka paljon pisteitä kummankin kategorian ominaisuuksiin voidaan sijoittaa.

Tapausnumero 111 merkitsee sitä, että mikään ei onnistunut edellisellä kierroksella. Säätämisessä keskitytään siis puolustuksen parantamiseen, ja ominaisuus, johon panostetaan eniten, on vihollisen nopeus. Tapausnumero 112 tarkoittaa sitä, että vihollinen on osunut pelaajaan hyvin, mutta selviytymisvalmiudet ovat olleet liian heikot. Tilannetta pyritään ensisijaisesti korjaamaan nostamalla aluksen nopeutta sekä tavoite-etäisyyttä. Tapausnumero 121 kertoo siitä, että vihollinen on kyennyt väistämään pelaajan ammuksia hyvin, mutta se ei ole selvinnyt tarpeeksi pitkään eikä se ole osunut pelaajaan tarpeeksi usein. Tällaisessa tilanteessa vihollisen tavoite-etäisyyttä lasketaan kaksi pykälää alemmaksi, jotta pelaajan olisi vaikeampi väistellä sitä kohti ammuttuja laukauksia. Tapausnumero 122 tarkoittaa sitä, että vihollinen on onnistunut väistelemisessä sekä osumisessa, mutta elinaika jäi liian lyhyeksi. Elinaikaa pyritään parantamaan suurella kolmen pykälän panssari-parannuksella.

Tapausnumero 211 tarkoittaa sitä, että vihollinen on selvinnyt tarpeeksi kauan pelaajaa vastaan, mutta se ei ole osunut tarpeeksi usein. Tarkkuutta pyritään ensisijaisesti parantamaan nostamalla ammusten liikkumisnopeutta kahdella kehityspisteellä. Lisäksi erikoisominaisuuden parantaminen voi vaikuttaa positiivisesti osumatarkkuuteen, jos käytössä oleva ase on kivääri tai haulikko. Tapausnumero 221 merkitsee sitä, että puolustus on kaikin puolin kunnossa. Sen lisäksi että osumatarkkuutta pyritään parantamaan ammusten nopeuttamisella, tavoite-etäisyyttä pienennetään hieman. Tapausnumero 212 on

todella lähellä optimitapausta. Koska selviytyminen ja osuminen ovat jo toivotulla tasolla, voidaan keskittyä vahingon tuottamisen ja väistövalmiuksien parantamiseen. Vahinkoa parannetaan kahdella kehityspisteellä ja tavoite-etäisyyttä siirretään vähän kauemmaksi. Tapausnumero 222 on paras mahdollinen tilanne, koska silloin kaikki tavoitteet ovat täyttyneet. Koska selviytyminen ja osuminen ovat toivotulla tasolla, vahinkoa voidaan parantaa kahdella pisteellä. Lisäksi kaikkiin muihin ominaisuuksiin tavoite-etäisyyttä lukuun ottamatta lisätään yksi piste.

Tietojen tutkimisen jälkeen aloitetaan vihollisobjektien säätäminen. Ennen säätämisprosessin aloittamista kerättyjen tietojen pohjalta muodostetaan tapausnumero. Tapausnumero päättää, miten jokaisen erän jälkeen käytettäväksi tulevat kehityspisteet jaetaan. Jokainen kehityspiste vastaa joko 10 %:n tai yhden mittayksikön parannusta vihollisaluksen ominaisuuden kokonaisarvoon. Huolimatta siitä päättääkö tekoäly priorisoida hyökkäyksen vai puolustuksen, toiseen ominaisuus-sarjaan sijoitetaan neljä ja toiseen kaksi kehityspistettä. Eli esimerkiksi jos tekoäly katsoo, että puolustus on kunnossa, vihollisobjektin asetta parannetaan neljällä kehityspisteellä ja puolustusta kahdella kehityspisteellä. Päätöksenteko on kuvattu kuvassa 6.



Kuva 6. Parannettavien ominaisuuksien valintapuu.

Parannettavista ominaisuuksista on otettava huomioon, että aseen kolmas ominaisuus kehittyy joka kerta riippumatta siitä, mikä tapausnumero on kyseessä. Lisäksi panssari kehittyy eri tavalla vain yhdessä tapauksessa kahdeksasta.

4.7 Pelialueen muokkaaminen

Testiohjelmassani pelialueen muokkaamisesta vastaa pelaajan vaikutuksesta riippumaton proseduraalisen generaation algoritmi. Algoritmin tehtävänä on lisätä pelialueelle sen olosuhteita muuttavia objekteja. Nämä objektit ovat yhtä paljon pelaajaa kuin tekoälyvastustajaa vastaan. Algoritmin toiminta on yksinkertainen. Ensin se valitsee pelialueelta koordinaatit käyttäen satunnaislukugeneraattoria. Sen jälkeen algoritmi valitsee kyseisiin koordinaatteihin sopivan objektin.

Valittu objekti voi olla yksi mahdollisista erikokoisista seinäobjekteista tai raidetykillä varustettu tykkitorni. Mahdolliset valittavat objektit lisätään listaan, josta algoritmi poimii yhden objektin satunnaisessa järjestyksessä. Listassa on aina enemmän objekteja kuin pelialueelle on tarkoitus lisätä. Jokaisella taistelukierroksella on oma objektikiintiö. Kiintiön suuruus kasvaa jokaisen selvitetyn kierroksen jälkeen.

Pelialueelle lisättävät seinäobjektit ovat muuten samanlaisia kuin pelialueen rajaavat seinät, mutta niillä on oma osumapistemäärä. Tämä tarkoittaa sitä, että ne tuhoutuvat, jos ne vastaanottavat tarpeeksi vahinkoa. Seinäobjektit ovat valmiiksi luotuja pysty- tai vaakasuorassa olevia suorakulmioita. Objektien koko ja muoto pidetään tässä vaiheessa mahdollisimman yksinkertaisena, jotta niiden lisääminen pelialueelle onnistuu mahdollisimman helposti.

Tykkitornit ovat pohjimmiltaan samanlaisia peliobjekteja kuin pelaaja ja vihollinen, mutta ne eivät voi liikkua. Tykkitornit eroavat muista peliobjekteista isomman koon ja erilaisen värityksen avulla. Tykkitornit ovat taistelun alussa toimettomia ja neutraaleja. Toimeton tila rikkoutuu, mikäli toinen taistelun osapuolista ampuu jotain pelialueen tykkitornia. Tykkitornit ovat kaikki samalla puolella, eli mikäli yksi tykkitorni kääntyy jotain osapuolta vastaan, muut seuraavat perässä. Mikäli molemmat osapuolet onnistuvat vahingoittamaan tykkitornia, ne ampuvat lähimmäistä mahdollista kohdetta.

5 Algoritmikuvaukset

Tässä luvussa selitetään miten testiohjelman kaksi tärkeintä algoritmia toimivat. Ensimmäinen algoritmi luo vihollisen jokaisen erän alussa, ja toinen muokkaa kenttää. Koodilistaukset ovat kuvankaappauksia luodusta ohjelmakoodista.

5.1 Vihollisia luova algoritmi

Listauksessa 1 esitetty funktio luo vihollisen ennen jokaista taistelua. Funktio tarvitsee toimiakseen alkavan erän numeron, jotta se löytää edellisen erän tiedot tutkittavaksi (rivi 62). Funktio aloittaa luomalla vihollisinstanssin ja asettamalla sen ennalta määriteltyyn pisteeseen pelialueella (rivit 64–65). Sen jälkeen ehtolause määrittää, lisätäänkö luotuun vihollisobjektiin puuttuvat skriptit perusasetuksilla vai kerättyjen tietojen mukaan muunnelluilla asetuksilla (rivit 67–86). Jos funktio ohjautuu käyttämään kerättyjä tietoja, se aloittaa hakemalla edellisen erän tiedot (rivit 75–78). Kun tiedot on haettu, haetaan skripti, joka päättää kerättyjen tietojen perusteella tapausnumeron (rivit 80–83). Lopuksi kutsutaan funktiota, joka päättää ja asettaa uudet ominaisuusarvot luotuun vihollisobjektiin (rivi 85).

```
62  public void CreateEnemy(int round)
63  {
64      enemy =
65          Instantiate( Resources.Load("Enemy"),
66                          new Vector3(0f, 1.130005f, 6.628702f),
67                              Quaternion.identity) as GameObject;
68
69      if (round == 1 || round == 4 || round == 7)
70      {
71          GameObject enemyWeapon = enemy.transform.Find("Weapon").gameObject;
72          enemyWeapon.AddComponent<ShootingControllerEnemy>();
73          enemy.AddComponent<Enemy>();
74      }
75      else
76      {
77          int prev_round = round - 1;
78
79          InformationPackage previousRound =
80              GameObject.Find("InformationCollectedFromRound" + prev_round)
81                  .GetComponent<InformationPackage>();
82
83          DecisionTree dT = gameObject.AddComponent<DecisionTree>();
84
85          int caseNumber =
86              dT.DetermineCase( previousRound.timeGoalMet,
87                                  previousRound.accuracyGoalMet,
88                                      previousRound.evasionGoalMet);
89
90          DetermineStats(enemy, caseNumber);
91      }
92  }
```

Listaus 1.

Listauksessa 2 esitetty algoritmi selvittää oikean tapausnumeron saavutettujen tavoitteiden pohjalta (rivit 10–35). Saavutettujen tavoitteiden Boolean arvot kerätystä datasta selvittää listauksessa 3 kuvattu funktio.

```
7   public int DetermineCase(bool timeGoalMet, bool accuracyGoalMet, bool evasionGoalMet)
8   {
9       int caseNumber = 0;
10
11      if (timeGoalMet == false)
12      {
13          if (evasionGoalMet == false)
14          {
15              if (accuracyGoalMet == false) caseNumber = 111;
16              else if (accuracyGoalMet) caseNumber = 112;
17          }
18          else if (evasionGoalMet)
19          {
20              if (accuracyGoalMet == false) caseNumber = 121;
21              else if (accuracyGoalMet) caseNumber = 122;
22          }
23      }
24      else if (timeGoalMet)
25      {
26          if (evasionGoalMet == false)
27          {
28              if (accuracyGoalMet == false) caseNumber = 211;
29              else if (accuracyGoalMet) caseNumber = 212;
30          }
31          else if (evasionGoalMet)
32          {
33              if (accuracyGoalMet == false) caseNumber = 221;
34              else if (accuracyGoalMet) caseNumber = 222;
35          }
36      }
37      return caseNumber;
38  }
```

Listaus 2.

Tapausnumero määrittää, mitä ominaisuuksia tulee parantaa ja kuinka paljon. Niiden kolmesta numerosta koostuva nimi määräytyy siten, että jokainen epäonnistunut tavoite merkitään numerolla 1 ja onnistunut tavoite numerolla 2. Kolmesta numerosta ensimmäinen merkitsee aikatavoitetta, toinen väistötavoitetta ja kolmas osumatavoitetta. Esimerkiksi jos väistötavoite on ainoa onnistunut tavoite, tapausnumeroksi tulee 121. Kun tapausnumero on saatu selvitettyä, se välitetään eteenpäin funktion palautusarvona (rivi 36).

```
61  void CompareToGoals()
62  {
63      if (time > 30.0f)timeGoalMet = true;
64      else timeGoalMet = false;
65
66      if (playerAccuracy < 0.25)evasionGoalMet = true;
67      else evasionGoalMet = false;
68
69      if (enemyAccuracy > 0.25)accuracyGoalMet = true;
70      else accuracyGoalMet = false;
71
72  }
```

Listaus 3.

Listauksessa 4 on esitelty funktio, joka välittää oikeat tiedot parannuksista ne toimeenpaneville skripteille. Funktio saa parametrina aiemmin luodun vihollisinstanssin sekä tapausnumeron (rivi 85). Funktion toiminta alkaa tarvittavien skriptien lisäämisellä vihollisobjektiin ja sen aseeseen (rivit 89–90). Lisäksi luodaan tarvittavat viittaukset lisättyihin skripteihin, jotta niiden sisältämiä funktioita voidaan kutsua prosessin aikana (rivit 92–94).

```
94   public void DetermineStats(GameObject enemy, int caseNumber)
95   {
96       GameObject enemyWeapon = enemy.transform.Find("Weapon").gameObject;
97
98       enemyWeapon.AddComponent<ShootingControllerEnemy>();
99
100      enemy.AddComponent<Enemy>();
101
102      Enemy ship = enemy.gameObject.GetComponent<Enemy>();
103
104      ShootingControllerEnemy weapon =
105          enemy.gameObject.GetComponentInChildren<ShootingControllerEnemy>();
106
107      switch (caseNumber)
108      {
109          case 111:
110                  AssignDefValues(1.1f, 1.2f, 1.1f);
111                  AssignAtkValues(1.0f, 1.1f, 1, 0.9f);
112          break;
113
114          case 121:
115                  AssignDefValues(1.1f, 1.1f, 0.8f);
116                  AssignAtkValues(1.0f, 1.1f, 1, 0.9f);
117          break;
```

Listaus 4.

Kun skriptit on lisätty ja viittaukset luotu, selvitetään tapausnumeron avulla millä tavalla ominaisuuksia pitää parantaa. Listauksessa 4 on esitetty kaksi kahdeksasta mahdolli-

sesta tapauksesta (rivit 98–106). Rivien 99–100 ja 104–105 funktiot esitellään tarkemmin listauksessa 10. Lisäksi listauksessa 5 on esitetty ns. oletustapaus, jota käytetään aina ensimmäisellä luontikerralla ja silloin, kun vihollisaluksen asetta vaihdetaan. Lopuksi funktio välittää tiedot luodulle vihollisobjektille käyttäen sen skripteissä olevia käsittelyfunktioita (Listaus 6, rivit 155–158).

```
149         default:
150                 calc_armor = default_armor;
151                 calc_speed = default_speed;
152                 calc_distance = default_distance;
153
154                 shotgun_rifle_calc_dmg = shotgun_rifle_default_dmg;
155                 shotgun_rifle_calc_spd = shotgun_rifle_default_spd;
156                 shotgun_rifle_calc_ammoB = shotgun_rifle_default_ammoB;
157
158                 railgun_calc_dmg = railgun_default_dmg;
159                 railgun_calc_spd = railgun_default_spd;
160                 railgun_calc_cldwn = railgun_default_cldwn;
161         break;
162     }
163     weapon.CreateWeapons
164         (shotgun_rifle_calc_dmg, shotgun_rifle_calc_spd, shotgun_rifle_calc_ammoB,
165             railgun_calc_dmg, railgun_calc_spd, railgun_calc_cldwn);
166
167     ship.CreateShip(calc_armor, calc_speed, calc_distance);
168
169     RefreshPrevValues();
170 }
```

Listaus 5.

```
172 void AssignDefValues(float armorB, float speedB, float distanceM)
173 {
174     calc_armor = (float)prev_armor * armorB;
175     calc_speed = (float)prev_speed * speedB;
176     calc_distance = (float)prev_distance * distanceM;
177 }
178
179 void AssignAtkValues(float damageB, float speedB, int ammoB, float cldwnM)
180 {
181     shotgun_rifle_calc_dmg = (float)shotgun_rifle_prev_dmg * damageB;
182     shotgun_rifle_calc_spd = (float)shotgun_rifle_prev_spd * speedB;
183     shotgun_rifle_calc_ammoB = shotgun_rifle_prev_ammoB + ammoB;
184
185     railgun_calc_dmg = (float)railgun_prev_dmg * damageB;
186     railgun_calc_spd = (float)railgun_prev_spd * speedB;
187     railgun_calc_cldwn = (float)railgun_prev_cldwn * cldwnM;
188 }
```

Listaus 6.

5.2 Kenttää muokkaava algoritmi

Tämä luku kuvaa tarkemmin kentänmuokkausalgoritmia. Algoritmin ohjelmakoodi on esitetty listauksissa 7–10. Listauksesta 7 alkaen esitetty kenttää muokkaavan algoritmin kooditiedosto alkaa tarvittavien kirjastojen lisäämisellä. Kaksi ensimmäistä kirjastoa lisätään automaattisesti jokaiseen kooditiedostoon pelimoottorin toimesta, mutta algoritmi tarvitsee toimiakseen myös System.Collections.Generic-kirjaston (rivit 1–3). Se sisältää kaiken tarvittavan lista-olioiden luomiseen, joita käytetään tietojen säilyttämiseen. Lisäksi algoritmi tarvitsee toimiakseen kolme muuttujaa, joihin tallennetaan objektien lukumäärät (rivit 7–9).

Ensiksi algoritmi laskee, kuinka monta objektia pelialueelle pitää lisätä alkavalla kierroksella. Funktio tarvitsee toimiakseen alkavan kierroksen numeron, joka välitetään funktiolle parametrina (rivi 11). Tykkitorneja lisätään yksi kappale jokaisen parillisen kierroksen jälkeen. Tämä varmistetaan jakamalla kierrosnumero kahdella ja pyöristämällä alaspäin käyttäen Unityn matematiikkakirjaston Floor-funktiota (rivit 13–14). Esimerkiksi jos alkava kierros on kolmas, tykkitorneja lisätään 3/2 + (Pyöristys alaspäin) = 1. Seinien määrä on kolme + kierroksen numero, eli ensimmäisellä kierroksella seiniä lisätään neljä ja viidennellä kahdeksan kappaletta (rivit 16–17). Lisäksi algoritmi laskee objektien yhteismäärän, jota tarvitaan apuna objekti-instanssien luomisessa.

```
1   using UnityEngine;
2   using System.Collections;
3   using System.Collections.Generic;
4
5   public class TerrainModifier : MonoBehaviour {
6
7       int towerAmount;
8       int wallAmount;
9       int objectAmount;
10
11      void CountObjectAmounts(int round)
12      {
13          towerAmount = round/2;
14          towerAmount = (int)Mathf.Floor(towerAmount);
15
16          wallAmount = 3+ round;
17
18          objectAmount = towerAmount + wallAmount;
19      }
```

Listaus 7.

Kun lisättävien objektien määrät on laskettu, algoritmin pitää lisätä seinät ja tykkitornit lista-objektiin odottamaan kentälle lisäämistä. Listauksen 8 funktio aloittaa luomalla tarvitun lista-objektin (rivi 24). Funktio lisää objektit listaan käyttäen toistorakenteita (rivi 26 ja 35). Rakenteiden apuna ovat funktiolle parametrina välitetyt objektimäärät (rivi 22).

Ensimmäinen toistorakenne aloittaa valitsemalla luvun yhden ja kolmen väliltä (rivi 28). Valittu luku määrittää, millainen seinäobjekti lisätään objektilistaan (rivit 30–32). Toistorakennetta toistetaan kunnes parametrina saatu seinäobjektien kokonaismäärä tulee täyteen. Kun seinät on lisätty listaan, on tykkitornien vuoro. Toinen toistorakenne on paljon yksinkertaisempi kuin ensimmäinen, koska tykkitorneja on vain yhdenlaisia (rivi 37). Lopuksi funktio palauttaa valmiin objektilistan (rivi 39).

```
21  List <string> AddObjectsToList(int wallAmount, int towerAmount)
22  {
23      List <string> objectList = new List<string>();
24
25      for (int i = 0; i < wallAmount; i++)
26      {
27          int rng = Random.Range(1,4);
28
29          if (rng == 1)objectList.Add("HorizontalWall");
30          if (rng == 2)objectList.Add("VerticalWall");
31          if (rng == 3)objectList.Add("SquareWall");
32      }
33
34      for (int i = 0; i < towerAmount; i++)
35      {
36          objectList.Add("Turret");
37      }
38      return objectList;
39  }
```

Listaus 8.

Kun objektilista on valmis, listauksessa 9 esitetty funktio laskee niille paikkakoordinaatit. Funktio aloittaa luomalla kolmiulotteisen vektorin, jota käytetään koordinaattien lisäämisessä lista-objektiin (rivit 44–45). Itse koordinaattien laskeminen hoidetaan käyttämällä satunnaislukugeneraattoria. Toistorakenteen sisällä käytettävä generaattori valitsee pelialueen sisältä satunnaisesti x- ja z-koordinaatit, johon objekti voidaan myöhemmin lisätä (rivit 49–50). Y-koordinaatti on vakio, koska kaikkien objektien tulee olla samalla korkeudella ohjelman toiminnan varmistamiseksi (rivi 51). Kun koordinaatit on

laskettu, muodostunut vektori lisätään lista-objektiin (rivi 53). Lopuksi funktio palauttaa koordinaatit sisältävän listan (rivi 55).

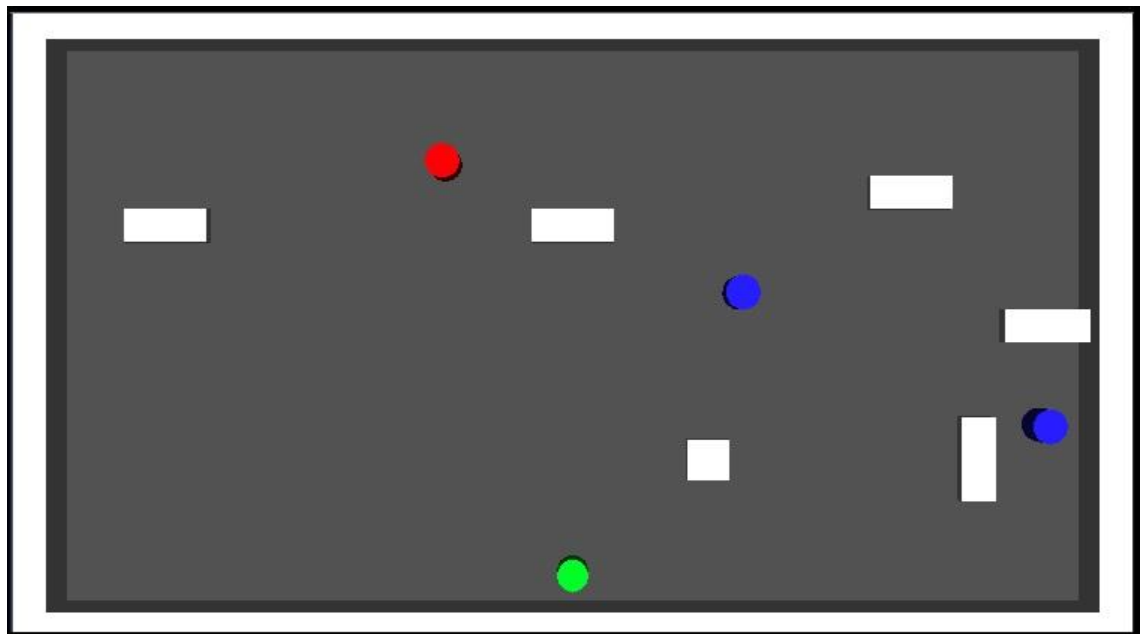
```
41  List <Vector3> ListLocations(int objectAmount)
42  {
43      Vector3 location;
44      List<Vector3> locationList = new List<Vector3>();
45
46      for (int i = 0; i < objectAmount; i++)
47      {
48          location.x = Random.Range(-15, 15);
49          location.z = Random.Range(-8, 8);
50          location.y = 1.130005f;
51
52          locationList.Add(location);
53      }
54      return locationList;
55  }
```

Listaus 9.

Listauksessa 10 on algoritmin toiminnan kannalta tärkein funktio. Se käyttää aiemmin esiteltyjä funktioita, jotta kentän muokkaaminen saadaan toteutettua. Ensin funktio laskee, montako objektia pelialueelle on lisättävä (rivi 60). Sen jälkeen se käyttää funktioita, jotka muodostavat tarvittavat tietolistat (rivit 61–62). Lopuksi funktio käyttää toistorakennetta objektien kentälle lisäämiseen (rivi 66). Toistorakenteen läpikäynnin aikana sen sisälle sijoitetut ehtolauseet tarkistavat, mikä objekti on seuraavana lisäysjonossa (rivit 68, 71, 74 ja 77). Kun objektin tyyppi on selvinnyt, se lisätään pelialueelle (rivit 69, 72, 75 ja 78). Kuvassa 7 on esitetty yksi mahdollinen algoritmin muokkaama pelialue.

```
57  public void ModifyTerrain(int round)
58  {
59      CountObjectAmounts(round);
60      List <string> objectList = AddObjectsToList(wallAmount, towerAmount);
61      List <Vector3> locationList = ListLocations(objectAmount);
62
63      int i = 0;
64
65      foreach (Vector3 location in locationList)
66      {
67          if (objectList[i] == "HorizontalWall")
68              Instantiate(Resources.Load("HorizontalWall"),
69                  location, Quaternion.identity);
70
71          if (objectList[i] == "VerticalWall")
72              Instantiate(Resources.Load("VerticalWall"),
73                  location, Quaternion.identity);
74
75          if (objectList[i] == "SquareWall")
76              Instantiate(Resources.Load("SquareWall"),
77                  location, Quaternion.identity);
78
79          if (objectList[i] == "Turret")
80              Instantiate(Resources.Load("Turret"),
81                  location, Quaternion.identity);
82
83          i++;
84      }
85  }
```

Listaus 10.



Kuva 7. Esimerkkitapaus kierroksen 3 kentästä.

6 Toimivuuden testaus

Opinnäytetyön kuudes luku käsittelee käytännön osuudessa toteutetun testiohjelman toimivuuden testaamisen. Aloitan kertomalla itse testistä, jonka jälkeen käyn läpi sen aikana kerätyt tiedot, ja lopuksi analysoin testin tuloksia.

6.1 Testin suunnittelu

Ohjelman toimivuuden testaamiseksi päätin tehdä erittäin pienimuotoisen ensimmäisen osapuolen testin. Testissä pelaan testiohjelman läpi kolme kertaa, valiten joka kerta eri aseen ja ominaisuusbonukset eri järjestyksessä. Testin tavoitteena on selvittää, onnistuuko algoritmi luomaan toisistaan eroavia vihollisia. Toimivuutta voidaan arvioida sillä kuinka, paljon lopputulokset eroavat toisistaan. Kolmen pelikerran alkuasetelmat on kuvattu taulukossa 9 ja pelikertojen tulokset on kuvattu taulukoissa 10-12.

Taulukko 9. Testikierrosten lähtökohdat.

Testikierros	Ase	Vahinko (+bonus)	Kestävyys (+bonus)	Nopeus (+bonus)
1	Haulikko	10 (+10 %)	100 (+20 %)	6 (+75 %)
2	Konekivääri	10 (+30 %)	100 (+40 %)	6 (+25 %)
3	Raidetykki	25 (+ 20 %)	100 (+60 %)	6 (+50 %)

Ominaisuuksien priorisoimisessa on otettu huomioon jokaisen aseen vahvuudet ja heikkoudet. Haulikkoa käyttäessä pitää pystyä lähestymään vihollisalusta, joten nopeus ja kestävyys ovat vahinkoa tärkeämpiä. Konekiväärin tapauksessa vahinko on tärkeämpää, koska sillä on helpoin ampua ohi. Kestävyys on nopeutta tärkeämpi tässä tapauksessa, koska konekivääriä käyttäessä ei välttämättä tarvitse päästä yhtä lähelle kuin haulikon kanssa. Raidetykin tapauksessa kestävyys on tärkeintä, koska aseella ei voi ampua useasti peräkkäin, eli vihollisalus voi ampua pelaajaa kohti useasti sillä välin kun pelaajan ase lataa. Vahinko on raidetykkiä käytettäessä tärkeämpää kuin nopeus, koska sitä käyttäessä on turvallisempaa pysyä mahdollisimman kaukana vihollisesta ja osua mahdollisimman usein.

Aluksi on huomioitava, että vihollisia parantavan algoritmin nykyinen versio kehittää aseen erityisominaisuutta saman verran tapausnumerosta riippumatta. Panssari puolestaan kehittyy saman verran joka kerta, paitsi sellaisen kierroksen jälkeen, jolloin ainoa epäonnistunut tavoite on aikatavoite. Yksikään kierros kolmen pelikerran aikana ei päättynyt tällaiseen lopputulokseen, joten jokaisen vihollisen panssari kehittyi täysin samalla tavalla joka kerta.

6.2 Testin toteutus

Taulukko 10. Kivääriviholisen kehittyminen pelikerroilla 1-3.*

Erä	Tapaus-numero	Panssari	Nopeus	Etäisyys	Vahinko	Ammusten nopeus	Ammusten määrä
Pelikerta 1: Haulikko							
1	-	100	6	10	10	50	5
2	221	110	6	9	11	60	6
3	221	121	6	8	12,1	72	7
10	121	133,1	6,6	6,5	12,1	79,2	8
Pelikerta 2: Kivääri							
2	221	110	6	9	11	60	6
3	211	121	6,6	9	12,1	72	7
10	112	133,1	7,9	9,9	13,3	72	8
Pelikerta 3: Raidetykki							
2	221	110	6	9	11	60	6
3	111	121	7,2	9,9	11	66	7
10	211	133,1	7,9	9,9	12,1	79,2	8

*Taulukossa esitetyt arvot on pyöristetty yhden kymmenyksen tarkkuudelle.

Ensimmäinen taulukko kuvaa kivääriviholisen kehittymistä kolmen eri pelikerran aikana. Ensimmäisellä pelikerralla algoritmi keskittyi pienentämään vihollisen tavoite-etäisyyttä ja nostamaan ammusten nopeutta. Tavoite-etäisyyttä tiputettiin kolmen kierroksen aikana yhteensä 35 % ja ammusten nopeutta nostettiin yhteensä 58,4 %. Lisäksi

kolmen kierroksen aikana nostettiin vihollisen nopeutta yhteensä 10 % ja vahinkoa parannettiin 21,1 %.

Toisella pelikerralla vihollisen nopeutta nostettiin yhteensä 31,7 %, joka on selvästi enemmän kuin ensimmäisellä pelikerralla. Myös etäisyyden kanssa päädyttiin erilaiseen lopputulokseen. Ensimmäisen kierroksen jälkeen etäisyyttä laskettiin, mutta muutos kumottiin toisen kierroksen jälkeen, ja lopullinen tavoite-etäisyys oli käytännössä sama kuin alkuperäinen. Vihollisen aseen vahinkoa parannettiin toisella pelikerralla tasaisesti. Sitä nostettiin jokaisen kierroksen jälkeen 10 %, eli lopullinen vahinko oli 33 % suurempi. Aseen ammusten nopeutta kehitettiin kahteen otteeseen 20 %. Tehdyt parannukset auttoivat, sillä kolmannen kierroksen jälkeen ominaisuutta ei enää tarvinnut parantaa.

Kolmannella pelikerralla ammusten nopeuden parantamiseen käytettiin eniten kehityspisteitä. Vaikka sen kanssa päädyttiin samaan lopputulokseen kuin ensimmäisellä pelikerralla, muutokset tehtiin eri järjestyksessä. Toisin kuin ensimmäisellä pelikerralla, isoimmat parannukset ammusten nopeuteen tehtiin kierrosten 1 ja 3 jälkeen. Myös aseen vahingon kanssa päädyttiin samaan lopputulokseen kuin ensimmäisellä pelikerralla. Erona on, että tällä kertaa viimeinen muokkaus tehtiin vasta kierroksen 3 jälkeen. Vihollisen lopullinen nopeus ja tavoite-etäisyys olivat puolestaan samoja kuin toisella pelikerralla. Vaikka lopputulokset olivatkin samoja, niihin päädyttiin erilaisten välivaiheiden kautta. Toisin kuin toisella pelikierroksella nopeutta nostettiin ensin 20 % kierroksen 2 jälkeen, ja 10 % kierroksen 3 jälkeen. Parannukset tehtiin siis samassa vaiheessa, mutta päinvastaisessa järjestyksessä. Etäisyyden kohdalla ensimmäisen kierroksen aikana tehdyt muutokset kumottiin jo kierroksen 2 jälkeen, eikä vasta ennen viimeistä kierrosta kuten toisella pelikerralla.
Vaikka lähtökohdat ovat jokaisella pelikerralla samat, päädyttiin joka kerta vähintäänkin osittain erilaisiin lopputuloksiin. Ensimmäisellä pelikerralla viimeinen kivääriviholllinen pyrkii asettautumaan lähemmäksi pelaajaa kuin kahden jälkimmäisen pelikerran viimeiset viholliset. Lisäksi sen nopeus on muita kolmikon jäseniä heikompi. Toisella ja kolmannella pelikerralla lopulliset ominaisuudet eroavat vain vahingon ja ammusten nopeuden puolesta. Toisen vihollisen ammukset tekevät enemmän vahinkoa, mutta kol-

mannen vihollisen ammukset ovat nopeampia, eli sen tarkkuus on todennäköisesti parempi.

Taulukko 11. Haulikkovihollisen kehittyminen pelikerroilla 1-3.*

Erä	Tapaus-numero	Panssari	Nopeus	Etäisyys	Vahinko	Ammusten nopeus	Ammusten määrä
Pelikerta 1: Haulikko							
4	-	100	6	10	10	50	5
5	121	110	6,6	8	10	55	6
6	221	121	6,6	7,2	11	66	7
10	121	133,1	7,3	5,8	11	72,6	8
Pelikerta 2: Kivääri							
5	221	110	6	9	11	60	6
6	221	121	6	8,1	12,1	72	7
10	111	133,1	7,2	8,9	12,1	79,2	8
Pelikerta 3: Raidetykki							
5	221	110	6	9	11	60	6
6	221	121	6	8,1	12,1	72	7
10	111	133,1	7,2	8,9	12,1	79,2	8

*Taulukossa esitetyt arvot on pyöristetty yhden kymmenyksen tarkkuudelle.

Taulukko 11 kuvaa pelin toista vastustajaa eli haulikkovihollista. Ensimmäisellä pelikerralla suurimmat muutokset tehtiin tavoite-etäisyyteen ja ammusten nopeuteen. Vihollisen tavoite-etäisyyttä laskettiin jokaisen kierroksen jälkeen. Ensimmäisen kierroksen jälkeinen pudotus oli 20 %, toisen 10 % ja vielä lopuksi toisen kerran 20 %. Viimeisen vihollisen tavoite-etäisyys oli jopa 42 % lähempänä pelaajaa kuin alkuperäinen tavoite. Ammusten nopeutta kehitettiin myös jokaisen kierroksen jälkeen. Viimeisen vihollisen aseen ammukset kulkivat 45,2 % nopeampaa kuin ensimmäisellä kierroksella. Aluksen nopeutta nostettiin vain kahteen otteeseen eikä lopullinen nopeusarvo ollut kuin 22 % suurempi kuin alkuperäinen. Vihollisen aseen vahinkoa puolestaan parannettiin ainoastaan 10 %.

Haulikkovihollinen kehittyi täysin saman kaavan mukaisesti toisella ja kolmannella pelikerralla. Aluksen nopeutta kehitettiin vasta viimeiselle kierrokselle, ja lopullinen nopeus oli periaatteessa sama kuin ensimmäisellä kierroksella. Tavoite-etäisyyttä muokattiin jokaisen kierroksen jälkeen, mutta toisin kuin ensimmäisellä pelikerralla, etäisyyttä ei ainoastaan pienennetty. Etäisyyttä laskettiin 10 % kahteen otteeseen, mutta jälkimmäinen pudotus kumottiin kolmannen kierroksen jälkeen. Vihollisen aseen vahinkoa nostettiin kahteen otteeseen. Molemmilla kerroilla vahinkoa parannettiin 10 %, eli toisen ja kolmannen pelikerran lopullinen vahinko oli vähän suurempi kuin ensimmäisellä pelikerralla. Aseen ammusten nopeutta kehitettiin täsmälleen samalla tavalla kuin ensimmäisellä pelikerralla.

Haulikkovihollinen kehittyi joka kerta todella samankaltaisesti. Ensinnäkin toisen ja kolmannen pelikerran viholliset kehittyivät täysin identtisesti. Sen lisäksi ammusten nopeus kehittyi joka kerta täysin samalla tavalla, ja itse vihollisien nopeuksissa on vain erittäin pieni ero. Suurin ero puolestaan saatiin tavoite-etäisyydessä. Ensimmäisellä pelikerralla tavoite-etäisyyttä pienennettiin selvästi, mutta jälkimmäisillä kerroilla ero alkuperäiseen on pieni.

Taulukko 12. Raidetykkivihollisen kehittyminen pelikerroilla 1-3.*

Erä	Tapaus-numero	Panssari	Nopeus	Etäisyys	Vahinko	Ammusten nopeus	Latausväli
Pelikerta 1: Haulikko							
7	-	100	6	10	25	65	2,5
8	221	110	6	9	27,5	78	2,3
9	111	121	7,2	9,9	27,5	85,8	2
10	121	133,1	7,9	7,9	30,3	94,4	1,8
Pelikerta 2: Kivääri							
8	221	110	6	9	27,5	78	2,3
9	111	121	7,2	9,9	27,5	85,8	2
10	221	133,1	7,2	8,9	30,3	102,7	1,8
Pelikerta 3: Raidetykki							
8	111	110	7,2	11	25	71,5	2,3
9	111	121	8,6	12,1	25	78,65	2
10	221	133,1	8,6	10,9	27,5	94,4	1,8

*Taulukossa esitetyt arvot on pyöristetty yhden kymmenyksen tarkkuudelle.

Ensimmäisellä pelikerralla suurimmat muutoskohteet olivat etäisyys ja ammusten nopeus. Tavoite-etäisyyttä pienennettiin ensin hiukan, mutta muutos kumottiin toisen kierroksen jälkeen. Kolmannen kierroksen jälkeen tavoite-etäisyyttä tiputettiin vielä alemmaksi kuin se aiemmin oli. Viimeisen kierroksen 21 % pienempi tavoite-etäisyys on pienempi kuin muilla pelikerroilla säädetyt etäisyydet. Ensimmäisellä pelikerralla ohjelma panosti siihen, että pelaajan olisi mahdollisimman vaikea väistellä ammuksia. Lyhyimmän tavoite-etäisyyden lisäksi siihen viittaa jokaisen kierroksen jälkeen kehitetty ammusten nopeus. Lisäksi itse aluksen nopeutta ja aseen vahinkoa kehitettiin yhteensä kahteen otteeseen. Nopeutta kehitettiin 32 % ja vahinkoa 21 %.

Toisella pelikerralla ammusten nopeus oli eniten kehitetty ominaisuus. Sitä kehitettiin joka kerta, ja 20 % korotuksia tehtiin ensimmäisen ja kolmannen kierroksen jälkeen. Ammusten lopullinen nopeus oli jopa 58 % alkuperäistä arvoa suurempi. Tavoite-etäisyyttä muokattiin samalla tavalla kuin ensimmäisellä pelikerralla, mutta viimeinen pienennys oli puolet pienempi. Vihollisen aluksen nopeutta parannettiin vain toisen

kierroksen jälkeen, jolloin sitä kasvatettiin 20 %. Aseen vahinkoa puolestaan kehitettiin samalla tavalla kuin ensimmäisellä pelikerralla.

Kolmannella pelikerralla vihollista kehitettiin päinvastaiseen suuntaan. Ensinnäkin nopeutta oli nostettu kolmanteen kierrokseen mennessä 43 %. Kumpikaan edellisistä vihollisista ei saavuttanut samaa nopeutta kolmannen kierroksen jälkeen. Kolmannen pelikerran raidetykkivihollinen on ainoa, joka päätti nostaa tavoite-etäisyyttä alkuperäistä tasoa korkeammaksi. Kierroksella kolme tavoite-etäisyys oli 21 % korkeampi kuin ensimmäisellä kierroksella. Etäisyyttä kumminkin pudotettiin hieman viimeiselle kierrokselle, jossa tavoite-etäisyys oli 9 % kauempana alkuperäisestä. Kolmannella pelikerralla vahingon kehittäminen oli pienin prioriteetti. Sitä parannettiin vain 10 % juuri ennen viimeistä kierrosta. Kolmannella pelikerralla päädyttiin samaan lopputulokseen ammusten nopeuden osalta kuin ensimmäisellä pelikerralla. Erona oli, että ensimmäisellä pelikerralla suurempi parannus tehtiin ensimmäisen kierroksen jälkeen. Kolmannella pelikerralla se tehtiin vasta kolmannen kierroksen jälkeen.

6.3 Tulosten analysointi

Taulukko 13. Tapausnumeroiden ilmenemiskerrat testin aikana.

Tapausnumero	**Ilmenemiskerrat**
221	13
111	7
121	4
211	2
112	1
122	0
212	0
222	0
Kierrokset:	**27**

Kuten taulukosta 13 näkee, selvästi yleisin tapausnumero oli 221. Tapauksen yleisyys voi johtua esimerkiksi tähtäämisen vaikeudesta tai pelaajan yleisestä taidottomuudesta. Jos katsotaan, että tietty tapausnumero ilmenee liian usein, asiaan voidaan vaikuttaa

mm. pelimekaniikkoja parantamalla tai tavoitteita nostamalla. Tavoitteiden nostaminen on ohjelmoinnin kannalta helpoin keino vaikuttaa tavoitteen täyttymisiin, mutta jos tavoitteista tehdään tekoälylle liian haastavat, vihollisien säätämisestä katoaa selvä logiikka. Asiaa pitää ajatella siitä näkökulmasta, että nämä tavoitteet ovat tekoälyn itselleen asettamia, eikä siinä olisi mitään järkeä, että se asettaisi itselleen mahdottomia tavoitteita.

Toiseksi yleisin tapausnumero 111 puolestaan viestii siitä ettei mikään onnistunut. Tavoitteen ilmenemiseen ei tässä vaiheessa kannata puuttua. Testiohjelman nykyisessä versiossa kentälle lisättävät tykkitornit voivat joissain tapauksissa tehdä selvää vihollisesta, ennen kuin kierros pääsee kunnolla alkamaan. Tekoäly ei tässä vaiheessa huomioida tykkitorneja tähdätessään, joten ne ovat suurempi uhka tekoälylle kuin pelaajalle.

Vihollinen onnistui väistötavoitteessa 63 % kierroksista, ja epäonnistui osumatavoitteessa 96 % kierroksista. Tästä voidaan päätellä, että nykyisessä versiossa osuminen on vaikeaa. Koska tulokset ovat niin selviä, tällaisessa tilanteessa pitää pohtia, miten osumista voisi helpottaa. Ominaisuuksien lähtökohtien nostamisen lisäksi ammusten koon kasvattaminen parantaa helpottaa osumista. Pelaajan kohdalla osumatarkkuuteen voi myös vaikuttaa ohjainlaite. Jotkut pelaajat voivat pelata paremmin, jos käytössä on hiiri ja näppäimistö. Testiohjelman nykyinen versio tukee ainoastaan peliohjainta.

7 Pohdinta

Opinnäytetyön viimeinen luku sisältää testiohjelman konseptin ja toteutuksen arvioinnin. Lisäksi esittelen muutamia jatkokehitysideoita ja teen lopullisen loppuyhteenvedon.

7.1 Idean arviointi

Opinnäytetyön teoriaosuudessa käyn läpi esimerkkejä hyvin ja huonosti toteutetusta dynaamisen vaikeustason säätelystä. Koostin esimerkkien pohjalta taulukon johon on

listattu ominaisuuksia joita dynaamisella vaikeustasolla kannattaa ja ei kannata olla. Ominaisuudet löytyvät taulukosta 14, ja ne selitetään tarkemmin luvussa 2.3.

Taulukko 14. Dynaamisen vaikeustason säätelyn ominaisuuksien toteutuminen testiohjelmassa.

Tee näin	**Toteutuminen testiohjelmassa**
Tee ominaisuudesta vapaaehtoinen	Ei
Piilota ominaisuuden olemassaolo	Ei
Älä rankaise pelaajaa	Kyllä
Älä mahdollista epäloogisuuksia	Kyllä
Älä luo liian vahvaa kuminauhaefektiä	Kyllä

Koska testiohjelman idea on, että dynaaminen vaikeustaso luo itse peliä, ominaisuutta ei voi säätää vapaaehtoiseksi. Dynaaminen vaikeustaso on erittäin suuri osa itse peliä, joten myöskään ominaisuuden piilottaminen ei todennäköisesti onnistu. Vaikka kumpikaan näistä hyväksi todetuista ominaisuuksista ei toteudu pelikonseptissa, en koe sitä ongelmaksi.

Ennalta hyväksi todettujen ominaisuuksien puutetta voi korvata keskittymällä huonojen ominaisuuksien poistamiseen. Yksi pelikonseptin kulmakivistä on, että pelin vastus on vahva mutta reilu. Jos tätä periaatetta seurataan mahdollisessa jatkokehityksessä, pelaajaa rankaisevaa dynaamista säätelyä ei pitäisi syntyä jatkossakaan. Sama asia pätee epäloogisuuksiin. Niitä ei ehdi muodostua testiohjelman nykyisessä versiossa, ja niiden muodostuminen jatkossa voidaan estää huolellisella testaamisella. Dynaamiseen vaikeustasoon kuuluu pelin vaikeuttamisen lisäksi pelin helpottaminen tarvittaessa.

Vaikka testiohjelman nykyinen versio ei sisällä vaikeustason laskemista, se pitää toteuttaa jossain vaiheessa mahdollista jatkokehitystä. Kuminauhaefektin muodostuminen pitää kuitenkin estää, koska se on täysin yhden pelin pääperiaatteen vastainen ilmiö. Pelin vastuksen on tarkoitus kasvaa koko ajan pelin kuluessa, joten vaikeustason helpottaminen ei ole hyväksyttävää. Vaikeustason nostamista voi kuitenkin väliaikaisesti hidastaa tai pysäyttää kokonaan. Nousuun voi vaikuttaa vähentämällä kehityspisteiden käyttöä. Vaikka kaikkia kehityspisteitä ei käytettäisi, niitä ei myöskään hukattaisi. Pisteet säästetään ja vaikeustasoa voidaan nostaa enemmän kun pelaajan suoritus paranee.

7.2 Jatkokehitysideoita

Opinnäytetyöprosessin aikana valmiiksi saamani ohjelma on vain supistettu osa siitä kokonaiskuvasta, mikä minulla on ollut. Tässä luvussa kerron ideoita pelin jatkokehitystä varten.

7.2.1 Vihollisalusten muokkaus

Testiohjelman tämänhetkisessä versiossa muuteltavia ominaisuuksia on kuusi kappaletta, jotka on jaettu kahteen eri kategoriaan: alukseen ja aseeseen. Jatkokehityksessä kategorioita voisi lisätä. Omia kategorioitaan voisivat olla esimerkiksi liikkuminen ja tähtääminen. Tällöin esimerkiksi alus-kategorian alle jäisi vain puolustukseen liittyviä ominaisuuksia ja kykyjä. Vihollisalusten kykyjä voisivat olla esimerkiksi törmäilyn mahdollistavat piikit, pelaajan liikettä ennakoiva tähtäys tai väistelyssä auttava pyrähdys. Tietenkin tällaiset kyvyt pitäisi lisätä myös pelaajan valittavaksi jossakin muodossa.

Toinen tapa, jolla voi luoda vaihtelua, on lisätä erilaisia ammustyyppejä. Esimerkiksi Borderlands-pelisarjassa on aseita jotka ampuvat esimerkiksi tuli-, sähkö- tai happoammuksia. Jokaisella ammustyypillä on oma heikkoutensa ja vahvuutensa. Uusien ammustyyppien rinnalle lisättäisiin vastaavat erikoispanssarit, jotka voivat neutralisoida erikoisammusten vaikutuksen. Pelaaja voisi löytää näitä erikoisammuksia tai -panssareita sattumanvaraisesti, ja vihollisten pitäisi varustaa omat aluksensa vastakeinoilla.

Viholliset voisivat myös oppia pelaajan käytöksestä. Esimerkiksi jos pelaajalla on tapana pysyä kaukana silloin kun ase lataa ja lähestyä vain silloin kun on tarkoitus ampua, viholliset voisivat oppia joko käyttämään tilannetta hyväksi tai puolustautumaan tehokkaammin. Tämän voisi toteuttaa suhteellisen yksinkertaisesti: verrataan etäisyyttä josta pelaaja ampuu etäisyyteen jossa pelaaja on silloin kun ase lataa. Jos etäisyyksissä on selvä ero, voidaan olettaa, että kun pelaaja lähestyy, hän aikoo ampua. Tilannetta voi käyttää hyväksi vaikkapa viivyttämällä vihollisaluksen ampumista, kunnes pelaaja on

lähempänä. Jos vihollisaluksen prioriteettina on pidempi selviytyminen, se voi perääntyä tunnistaessaan tilanteen muodostumisen.

Testiohjelman nykyisen version dynaaminen vaikeustason säätely toimii vain yhteen suuntaan. Vaikka pelin ideana on, että vaikeustasoa nostetaan, sitä voisi kehittää siten, että vihollisia ei kehitetä jos pelaaja alisuorittaa liikaa. Vihollisen ansaitsemia kehityspisteitä ei kuitenkaan luovuteta tai hukata, vaan niitä käytetään sitten enemmän kun pelaajan suoritustaso paranee. Vihollinen voi myös säästää pisteitä sellaisiin ominaisuuksiin joita se saisi yleensä avattua vasta myöhemmin pelissä. Vaikeustaso pysyisi hetkellisesti samana, mutta se kostautuisi ennemmin tai myöhemmin suurempana vaikeustason nousuna.

7.2.2 Kentän muokkaus

Testiohjelmassa kenttää muokataan lisäämällä peliobjekteja satunnaisesti valittuihin paikkoihin. Lisätyt objektit ovat puolueettomia, mutta pidemmälle kehitetyssä ohjelmassa voisi olla myös sellaisia kenttiä jotka ovat puolueellisia. Esimerkiksi pelin alussa kenttä voisi olla pelaajan puolella, mutta pelin edetessä kentät muuttuisivat ensin puolueettomiksi ja lopulta pelaajan vastaisiksi. Lisäksi ohjelmaa voisi laajentaa lisäämällä uusia erilaisia kenttäpohjia tai kenttään asetettavia objekteja.

Testiohjelman nykyisessä versiossa kenttään lisättäviä objekteja on kahta eri lajia: seiniä ja tykkitorneja. Mikäli peli-ideaa kehittäisi pidemmälle, kentälle lisättävien objektien valikoimaan voisi lisätä myös esimerkiksi miinoja ja vaarallista ainetta sisältäviä lammikoita. Miinat voivat kenen tahansa osapuolen puolella, mutta vaara-alueet ovat lähtökohtaisesti yhtä suuri uhka jokaiselle osapuolelle. Testiohjelman nykyisessä versiossa objektit lisätään kentälle sattumanvaraisesti valittuihin koordinaatteihin. Jatkokehityksessä koordinaattien valinnassa voisi käyttää apuna lämpökarttoja, jos objektien halutaan ensisijaisesti toimivan pelaajaa vastaan.

7.3 Yhteenveto

Opinnäytetyön ensimmäinen tutkimuskysymys oli, millainen vaikeustason säätely on toiminut. Luvussa 2 esiteltiin pelejä, joissa dynaamisen vaikeustason luomisessa on onnistuttu. Näiden pelien yhdistävä tekijä on ominaisuuden hienovaraisuus. Half-Life 2 ja Max Payne - peleissä ominaisuutta käytetään niin vähän ja niin pieniin asioihin ettei sitä todennäköisesti huomaa. Vaikka Left 4 Dead -pelisarjassa pelaaja huomaa, että jokainen pelikerta on erilainen, hän ei välttämättä tiedä, että vaikeustaso vaihtelee oman tai ryhmän suorituksen mukaan. Näitä pelejä yhdistää myös se, että pelit käyttävät perinteistä staattista vaikeustasoa määrittämään pelin tarjoamaa haastetta, ja dynaaminen säätely keskittyy toissijaisiin kohteisiin. Lisäksi on huomioitava, että Half-Life 2 ja Max Payne toimisivat suoraan ilman dynaamista säätelyä. Näistä peliesimerkeistä voi tehdä johtopäätöksen, että dynaaminen vaikeustaso toimii parhaiten, kun sitä käytetään täydentämään perinteistä staattista vaikeustasoa. Toinen mahdollisuus tietenkin on antaa pelaajan valita, haluaako hän dynaamista vaikeustasoa, kuten on tehty Lego Star Wars 2 -pelissä.

Toinen kysymys oli, millainen dynaaminen vaikeustason säätely ei ole toiminut. Luvussa 2 on esitelty kolme esimerkkiä joissa dynaamisen vaikeustason katsotaan olevan pelikokemusta häiritsevä tekijä. NHL- ja Burnout-pelisarjoja ja The Elder Scrolls IV: Oblivion pelejä yhdistää se, että dynaamista vaikeustason säätelyä on käytetty liikaa ja liian ilmiselvästi. NHL- ja Burnout-pelisarjojen tapauksissa dynaaminen vaikeustason säätely vähentää saavutuksentunnetta, mikä voi olla syy miksi pelaaja harrastaa videopelien pelaamista. Dynaamisen vaikeustason käyttäminen urheilu- ja autopeleissä luo ristiriidan pelin muiden osa-alueiden kanssa. Pelihahmojen ja autojen omien ominaisuuksien merkitys katoaa, kun dynaaminen vaikeustaso muokkaa niitä. TES IV: Oblivionin kohdalla puolestaan dynaaminen vaikeustaso luo epäloogisuuksia, joiden vuoksi pelaaja ei koe edistyvänsä pelissä. Tästä tapauksesta voi päätellä, että mitä enemmän dynaamista vaikeustasoa haluaa käyttää, sitä monipuolisempi ominaisuuden on oltava. Vaikka dynaaminen vaikeustaso luo pääasiallisen haasteen peliin, se tarvitsee mielestäni myös staattisia tavoitteita. Näiden tavoitteiden täyttyessä, peli voi esimerkiksi korvata osan vihollisista olennoilla, jotka näyttävät ja tuntuvat edellisiä vahvemmilta. Lisäksi kannattaa miettiä, olisiko mahdollista nostaa vihollisten lukumäärää, niiden vahvistamisen sijaan.

Viimeinen tutkimuskysymys oli, miten dynaamista vaikeustasoa tulisi käyttää. Aiemmin esitetyt esimerkit onnistuneesta dynaamisesta säätelystä viittaavat siihen, että dynaaminen säätely toimii todella hyvin, jos sen olemassaoloa ei huomaa. Toisaalta tämä tarkoittaa sitä, että dynaamista säätelyä ei voi käyttää liikaa, koska sen huomaaminen helpottuu. Edes Left 4 Dead -pelisarja ei luota täysin dynaamiseen vaikeustasoon, vaan pelaaja valitsee staattisen vaikeustason, ja dynaaminen säätely täydentää sen puutteita. Omassa peliprototyypissäni tarkoituksena on rakentaa koko peli dynaamisen vaikeustason ympärille, joten sen piilottaminen ei onnistu mitenkään. En usko sen kuitenkaan olevan ongelma, sillä tässä tapauksessa dynaaminen vaikeustaso on idean keskipisteessä. Joko idea toimii tai sitten projektilla ei ole tulevaisuutta. Jos dynaamista vaikeustasoa halutaan käyttää johonkin muuhun kuin toissijaisten ominaisuuksien säätelyyn, se pitää ottaa huomioon heti projektin alkuvaiheessa ja tehdä siitä keskeinen ominaisuus.

Lähteet

Adams, E. 2008. The Designer's Notebook: Difficulty Modes and Dynamic Difficulty Adjustment. http://www.gamasutra.com/view/feature/132061/the_designers_notebook_.php?print=1. 3.4.2014

Borderlands Wiki. 2013. HellFire (Borderlands 2)/Variant Chart. http://borderlands.wikia.com/wiki/HellFire_%28Borderlands_2%29/Variant_Chart. 10.10.2013.

Epic Games. 2013. Documentation. http://www.unrealengine.com/en/udk/documentation/. 19.9.2013.

Game Ontology. 2013. Dynamic Difficulty Adjustment. http://www.gameontology.com/index.php/Dynamic_Difficulty_Adjustment. 19.11.2013.

HF Boards. 2013. Do you think the NHL series has Ice Tilt?. http://hfboards.hockeysfuture.com/showthread.php?p=70787009. 21.3.2014.

Left 4 Dead Wiki. 2014. The Director. http://left4dead.wikia.com/wiki/The_Director. 29.4.2014.

Mojang. 2011. Minecraft. Mojang.

Nelson, M, Togelius, J & Shaker, N. 2013. Chapter 1 Introduction(DRAFT). http://pcgbook.com/wp-content/uploads/2013/08/Introduction.pdf. 7.10.2013.

Overkill Software. 2013. Payday 2. 505 Games.

Rose, M. 2013. It's official: XNA is Dead. http://www.gamasutra.com/view/news/185894/Its_official_XNA_is_dead.php. 19.9.2013.

Tolentino, J. 2008. Good Idea, Bad Idea: Dynamic difficulty Adjustment. http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml. 3.4.2014.

Unity Technologies. 2013. Unity Scripting. http://unity3d.com/unity/workflow/scripting. 19.9.2013.

VashGH. 2013. NHL13 Ice Tilt Mechanic - In-Depth Look. https://www.youtube.com/watch?v=KQ_OLBh9vNs. 21.3.2014