Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Tommi Välkki

Kävelyn oppiminen geneettisellä algoritmilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

18.11.2018

Tekijä Otsikko Sivumäärä Aika	Tommi Välkki Kävelyn oppiminen geneettisellä algoritmilla 30 sivua 18.11.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Insinöörityössä oli tarkoituksena perehtyä geneettisten algoritmien toimintaan ja ratkaista niiden avulla sopiva optimointiongelma. Aiheeksi valittiin nelijalkaisen kävelijän kävelyn koneoppiminen geneettisellä algoritmilla. Projekti toteutettiin käyttäen Unity 3D:tä kävelijän ja ympäristön luontiin, ja projektin ohjelmointi toteutettiin C#-kielellä.

Työssä perehdyttiin geneettisten algoritmien historiaan ja niiden yhteyteen evoluutioteoriaan. Evoluutioteoria on perusta geneettisille algoritmeille, joiden ongelmanratkaisumenetelminä käytettiin evoluutiosta tuttuja käsitteitä, kuten geeni, genomi, kromosomi, risteytys, mutaatio ja populaatio.

Työ aloitettiin tutkimalla, kuinka kävelijän sai Unityssa toteutettua. Kävelijän luonnin jälkeen sitä liikuttamaan ohjelmoitiin moottori. Moottori sisälsi muuttujat, joista lopulta muodostui geneettisen algoritmin muovattava kromosomi, eli yksilö ja yksi potentiaalinen ongelman ratkaisu. Geneettisen algoritmin toteutuksessa luotiin ensin alkupopulaatio, jonka yksilöt pisteytettiin ja sitten yhdisteltiin seuraavan populaation luomiseksi. Lisäksi uudet yksilöt altistettiin mahdolliselle mutaatiolle. Uuden populaation iteraatioita luotiin, kunnes tarpeeksi hyvä yksilö ilmaantui tai sukupolvien maksimimäärä tuli täyteen.

Käyttäjäkokemuksen parantamiseksi luotiin populaation parasta yksilöä seuraava kamera, jonka yläreunassa esitettiin simulaation metriikoita. Lisäksi luotiin alkuvalikko, josta käyttäjä pystyi asettamaan simulaatiolle raja-arvoja tai testisimulaatiolle simulaatiosta saatuja ratkaisuja.

Työssä tuli vastaan ongelmia, joista huolimatta projekti saatiin onnistuneesti päätökseen ja saatiin toimiva, geneettistä algoritmia hyödyntävä simulaattori kävelyn koneoppimiselle. Simulaattorin käyttöliittymä mahdollisti ohjelman helppokäyttöisyyden. Tämä projekti antaa ohjeistavan suunnan vastaavanlaisille projekteille, joita tehtäessä voi hyödyntää tässä työssä esitettyjä ongelmia ja kehityskohtia paremman toiminnan saavuttamiseksi.

Avainsanat	geneettinen algoritmi, koneoppiminen, tekoäly, Unity, kävelijä

Author Title Number of Pages Date	Tommi Välkki Learning to walk with genetic algorithm 30 pages 18 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The objective of this thesis was to familiarize oneself with genetic algorithms. Find out how they work and solve a suitable optimization problem using genetic algorithm. Machine learned walking for a quadruped walker with genetic algorithm was selected as a subject of this thesis. The project was implemented with walkers and environment created with Unity 3D and C# as the programming language.

The history of genetic algorithms and the connection to evolution theory was addressed in this thesis. Evolution theory is the basis of genetic algorithms for which concepts like gene, genome, chromosome, crossover, mutation and population were methods of the problem solving.

The project was started by studying how the walker could be created in Unity. A motor was programmed to move the walker after its creation method was solved. The motor included variables from which an individual, chromosome or potential solution for the problem was formed for the use of genetic algorithm. In the implementation of genetic algorithm, first thing was to create initial population. The individuals of this population were evaluated for their fitness and then recombined to create next population. In addition, the individuals of new population were exposed to a possible mutation. The iterations for creating new population were carried out until an individual with good enough fitness was emerged or the maximum number of generations were reached.

To improve user experience a camera was created to follow the current best individual and also metrics were presented on top of the screen. In addition, a start menu was programmed from which user could set limits for the simulation or input results for a test simulation environment. User interface enabled accessibility of the simulator.

Several problems were encountered in the project but nevertheless the project was completed successfully. As a result, a functional simulator using genetic algorithm was created. This project can be used as a basis for similar projects and make use for encountered problems in their development.

Keywords	genetic algorithm, machine learning, AI, Unity, walker

Sisällys

1 Johdanto

Insinöörityön tarkoituksena oli perehtyä geneettisten algoritmien käyttöön ja ratkaista niiden avulla aiheeseen soveltuva optimointiongelma. Aiheeksi valittiin kävelyn koneoppiminen nelijalkaiselle kävelijälle, jonka mallinnus toteutettiin Unity 3D -pelimoottorilla ja ohjelmointi C#-ohjelmointikielellä. Työn tavoite oli toimia oppimiskokemuksena sekä esimerkkinä vastaavanlaista projektia tekevälle.

Geneettiset algoritmit, myöhemmin GA, ja niiden käsitteet pohjautuvat evoluutioteoriaan, jonka mukaan vahvimmat yksilöt selviytyvät ja siirtävät hyvät ominaisuutensa jälkeläisilleen. Tässä kontekstissa termit populaatio, vanhemmat, kromosomi, genomi, geeni, risteytys ja mutaatio tulevat tutuiksi luonnon ilmiön keinoin mallinnettuna koneoppimisena ja tekoälynä.

Insinöörityöraportissa käydään ensin läpi evoluution taustoja ja yhteyttä geneettisiin algoritmeihin. Tätä seuraa projektissa tärkeänä työkaluna käytetyn Unity 3D:n esittely. Seuraavaksi esitellään projekti ja sen toteutus, minkä jälkeen käydään vielä läpi toteutuksen haasteet ja ongelmat. Lopuksi arvioidaan työn lopputuloksia ja mahdollisia kehityskohtia.

2 Geneettiset algoritmit

Luvussa käsitellään ensin evoluutioteoriaa, jonka käsitteisiin ja ideoihin geneettiset algoritmit perustuvat. Tämän jälkeen käydään läpi geneettisten algoritmien historiaa ja yhteyttä evoluutioon sekä muutamia esimerkkejä eri työvaiheiden menetelmistä.

2.1 Evoluutioteoria

Geneettisten algoritmien idea perustuu evoluutioteoriaan, jonka mukaan vahvimmat yksilöt selviävät ja välittävät ominaisuuksiaan seuraaville sukupolville. Luonnossa evoluutio on jatkuva tapahtuma, joka muokkaa eliölajin ominaisuuksia vastaamaan paremmin ympäristön vaatimuksia. Ominaisuuksien muokkautuminen tapahtuu lajin sisällä hitaasti

lukuisten sukupolvien aikana, ja ominaisuudet voivat olla niin fyysisiä kuin käyttäytymiseenkin liittyviä.

Sellaiset yksilöt, joiden ominaisuudet sopivat parhaiten ympäristöön, selviävät huonoja ominaisuuksia omaavia paremmin ja näin tuottavat enemmän jälkeläisiä. Hyvät ominaisuudet siirtyvät jälkeläisille, ja lopulta ominaisuudet yleistyvät populaation tasolla. Monimuotoisuus voi vähentyä, koska lisääntymiskumppania ei valita satunnaisesti vaan hyvien ominaisuuksien perusteella, jolloin sisäsiittoisuus lisääntyy. Siksi on hyvä, että geeneiltään heikommat yksilöt pääsevät satunnaisesti lisääntymään. Tämän avulla monimuotoisuus säilyy, eikä evolutiiviseen umpikujaan jouduta niin helposti. [1.]

Monimuotoisuutta ylläpitää myös mutaatio. Mutaatio on sellainen muutos yksilön geenissä, jota ei pysty jäljittämään perimän kautta kumpaankaan vanhempaan tai aiempiin sukupolviin. Siinä missä evoluution päämääränä on pitkällä aikavälillä muuttaa lajin ominaisuuksia yksilö kerrallaan optimaaliseksi ja lähtökohtaisesti parempaan, on mutaatio sattumanvaraista. Mutaatiossa ominaisuus voi harpata evoluutiota nopeammin parannettuun ominaisuuteen tai huonontaa ominaisuutta hyvinkin radikaalisti. Toisaalta mutaatio voi olla myös yksilön kannalta neutraali. Haitalliset mutaatiot häviävät populaatiosta lopulta luonnonvalinnan seurauksena. [1, luku 4.1.]

2.2 Toteutus tietokoneella

Vuonna 1975 John Holland esitti ensimmäisen kerran tuloksia 1960-luvulla aloittamastaan tutkimuksesta, jonka tavoitteena oli siirtää luonnossa tapahtuvaa sopeutumista ohjelmointiin. Hollandin populaatioperustainen algoritmi oli ensimmäinen laatuaan. Nykyajan geneettiset algoritmit ovat kehittyneet Hollandin alkuperäisestä määritelmästä, eikä se ole enää nykyään olennainen osa geneettisiä algoritmeja. Toisaalta GA:lle ei myöskään ole tiukkaa ja yleisesti hyväksyttyä määritelmää, jota tulisi ohjelmoinnissa noudattaa. [2, s. 18.]

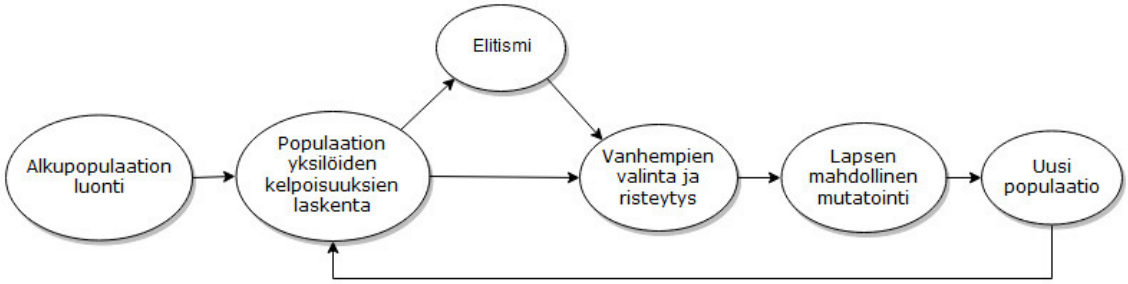
Geneettisissä algoritmeissa pyritään hyödyntämään evoluutioteorian käsitteistöä ratkaistaessa erilaisten asioiden optimointiongelmia. Keskeiset käsitteet optimointiongelman ratkaisemisessa geneettisellä algoritmilla ovat geeni, genomi, kromosomi, mutaatio, vanhemmat, risteytys, luonnonvalinta, alleeli ja populaatio.

Geneettisen algoritmin toiminta alkaa alkupopulaation satunnaisella luonnilla. Populaatio sisältää kaikki tutkittavat yksilöt, joista jokainen on erilainen. Populaation koon tulisi heijastaa ratkaistavan ongelman monimutkaisuutta, ja mitä enemmän geenejä kromosomissa on, sitä suurempi tulisi populaation koon olla. Populaation optimaalista kokoa voi silti olla vaikea määrittää, ja koon valintaan voivat vaikuttaa myös ulkoiset tekijät, kuten tietokoneen teho. Tarpeeksi suuri populaatio takaa populaation monimuotoisuuden, mutta toisaalta liian suuri populaatio hidastaa algoritmin toimintaa huomattavasti. [3, s. 18.]

Yksilöllä on erilaisia ominaisuuksia eli geenejä, ja nämä geenit yhdessä muodostavat genomin, joka on ohje yksilön muodostamiseksi, ja tämä ohje sijaitsee kromosomissa [4]. Käytännössä GA:ssa genomi on siis yksilö ja potentiaalinen ratkaisu ongelmaan. Geenin eri ilmentymät ovat biologiassa alleeleja, ja GA:ssa niitä vastaavat esimerkiksi binäärikoodauksen bittien tapauksessa 0 tai 1 ja arvoihin perustuvassa koodauksessa esimerkiksi liukuluvuissa kaikki annetun lukuvälin desimaaliarvot [5; 3, s. 13 ja 16.]

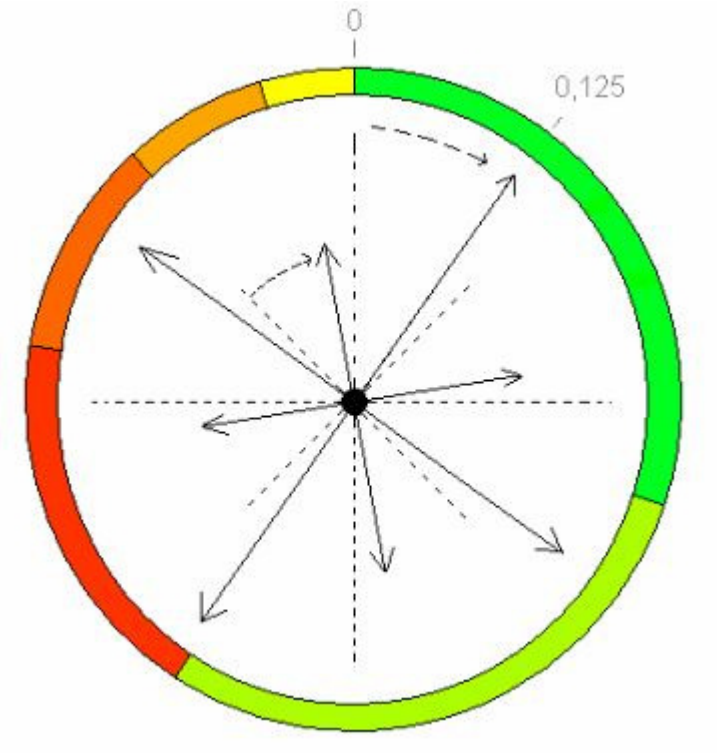
Evoluutiossa esiintyvä luonnonvalinta esitetään geneettisissä algoritmeissa yksilön kelpoisuutena optimointiongelman ratkaisuna. Ratkaisujen kelpoisuudet selvitetään pisteyttämällä kaikki populaation genomit. Kelpoisuuden laskenta on aina räätälöity ratkaistavaa ongelmaa varten, ja se on yleensä ainoa osa, jota ei pysty käyttämään suoraan muissa GA:lla ratkaistavissa ongelmissa, olettaen GA:n muiden osien olevan geneerisiä. Pisteytyksen jälkeen tarkistetaan, onko optimiratkaisu saavutettu, ja jos ei ole, genomit järjestetään kelpoisuusjärjestykseen seuraavan populaation risteytystä varten.

Uuden populaation luomiseen on monia erilaisia yhdisteltäviä tapoja, joista ensimmäinen on elitismi. Elitismissä tuodaan edellisestä populaatiosta tietty määrä parhaiten pisteytettyjä yksilöitä sellaisenaan uuteen populaatioon. Elitismi ei kuitenkaan ole pakollista, kuten kuvasta 1 nähdään.



Kuva 1. Geneettisen algoritmin vaiheet, joita jatketaan, kunnes tarpeeksi hyvä yksilö löytyy tai sukupolvien maksimimäärä on tullut täyteen.

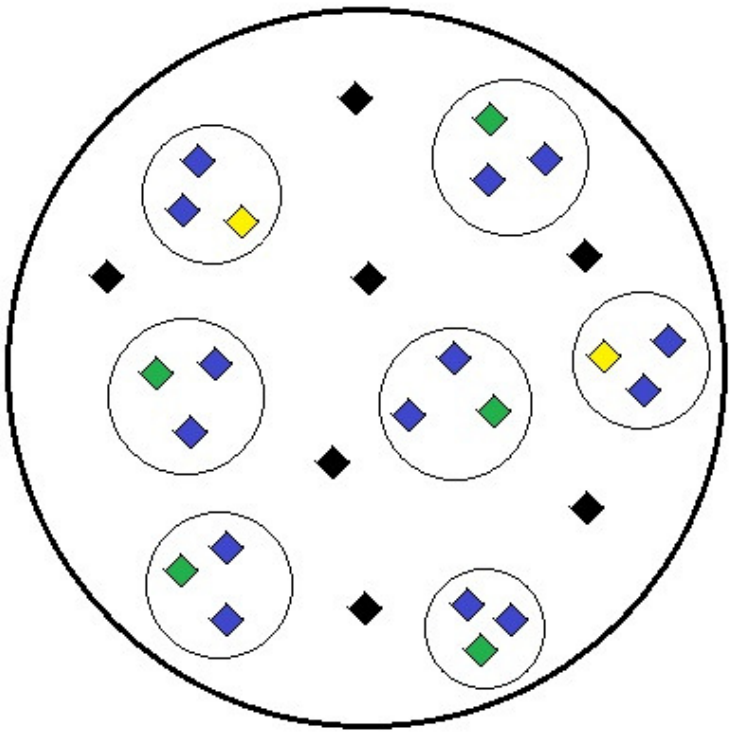
Mahdollisen elitismin jälkeen edellisen populaation yksilöitä risteytetään uusien genomien luomiseksi niin kauan, kunnes populaatio on täynnä. Uuden genomin vanhempien valintaan on useita vaihtoehtoja, esimerkiksi täysin satunnainen valinta. Satunnainen valinta ei painota yksilöiden kelpoisuutta millään tavalla, ja siksi sen käyttämistä vältetään GA:ssa. Rulettivalinta sen sijaan ei ole täysin satunnainen, vaan hyvien yksilöiden valintaa vanhemmiksi painotetaan niiden kelpoisuuden perusteella. Rulettivalinnassa on käytännössä osoitin, joka pysähtyy todennäköisimmin hyvän kelpoisuuden omaavan yksilön kohdalle. Stokastinen valinta on rulettivalinnan muunnelma, joka sisältää useita osoittimia, kuten kuvassa 2. Tällä tavalla voidaan valita useampi vanhempi kerrallaan ja monimuotoisuuden säilyminen saadaan varmistettua. Lisäksi hyvän kelpoisuuden omaava yksilö valitaan todennäköisesti ainakin kerran. [6.]



Kuva 2. Stokastinen valinta. Osoittimien määrä (tässä 8) määrää maksimiliukumaksi 1/8 = n. 0,125, joten liukuma on väliltä 0…0,125. Näin saadaan 8 vanhempaa valittua kerralla ja todennäköisyys on korkea hyvän kelpoisuuden omaavan vanhemman valinnalle. Populaation monimuotoisuus saadaan myös säilytettyä. [3, s. 20.]

Deterministisessä valinnassa paras tai parhaat yksilöt valitaan ensimmäiseksi vanhemmaksi ja toinen vanhempi valitaan populaatiosta satunnaisesti. Turnajaisvalinnassa populaatiosta valitaan satunnaisesti tietty määrä yksilöitä, esimerkiksi kolme yksilöä [6]. Joku näistä yksilöistä saattaa olla viimeinen populaation kelpoisuuksissa, toinen voi olla vähän parempi ja kolmas keskivaiheilla. Valinnan perustana on näistä kolmesta valita parhaan kelpoisuuden omaava yksilö vanhemmaksi, mikä painottaa parempien kelpoisuuksien valintaa. Kaikki valitut vanhemmat eivät kuitenkaan ole välttämättä kelpoisuudessaan huippuyksilöitä koko populaation tasolla.

Turnajaisvalintaa voi havainnollistaa esimerkillä jonkin eliölajin levinneisyydestä, kuten kuvassa 3, jossa populaation yksilöt ovat levittäytyneet laajalle alueelle. Yksilöt eivät hakeudu kaikki samalle pienelle alueelle lisääntymään, vaan valitsevat lisääntymiskumppaninsa oman ryhmänsä läheisyydessä olevista yksilöistä. Tämä ryhmä vastaa turnajaisvalinnan kolmea satunnaista yksilöä. Näin ollen lähellä olevista vaihtoehdoista paras yksilö pääsee jatkamaan sukuaan, vaikka se ei koko populaation tasolla olisi huippuyksilö.



Kuva 3. Populaatiossa lisääntyminen tapahtuu pienempien ryhmien sisällä, joissa ryhmän parhaat yksilöt (vihreät) ovat verrokkejaan (siniset) kelpoisempia jatkamaan sukuaan. Keltaiset yksilöt ovat populaation tasolla huippuyksilöitä, mutta niiden lisäksi vihreiden lisäännyttyä populaation monimuotoisuus säilyy.

Vanhemmat valittuaan GA:n on aika luoda lapsigenomi risteyttämällä, mihin on useita vaihtoehtoja. Muutamia risteytysvaihtoehtoja ovat yhden pisteen tekijäinvaihto, monen pisteen tekijäinvaihto ja satunnainen valinta. Yhden pisteen tekijäinvaihdossa genomista arvotaan satunnainen katkaisukohta, jonka alkupään geenit kopioidaan lapselle ensimmäiseltä vanhemmalta ja loppupuolisko toiselta vanhemmalta. Monen pisteen tekijäinvaihdossa katkaisukohtia on useita ja osat kopioidaan vuoroin kummaltakin vanhemmalta. Satunnaisessa valinnassa joka geenin kohdalla arvotaan, kummalta vanhemmalta geeni kopioidaan. Lapsen genomin määritettyään genomi vielä altistetaan mahdolliselle mutaatiolle, jonka tarkoituksena on estää populaation sisäsiittoisuutta. Mutaatiossa genomi käydään geeni geeniltä läpi ja mutaation todennäköisyydellä geeni korvataan uudella satunnaisella geenillä. [7.]

Populaation täytyttyä jokaisen yksilön kelpoisuus lasketaan jälleen, ja tätä jatketaan niin kauan, kuin tarpeeksi hyvä yksilö eli ratkaisu ongelmaan löytyy tai kunnes tietty määrä sukupolvia on käyty läpi.

Geneettinen algoritmi sopii erinomaisesti tarkasti rajatun, tarpeeksi yksinkertaisen ongelman ratkaisuun, kuten kävelyn opetus tasaisella alustalla tai reitin suunnittelu. Muita sovellusaloja ovat muun muassa robotiikka, biologia ja salaustekniikka. Dynaamiset ja hyvin monimutkaiset ongelmat, kuten kävelyn opetus muuttuvassa ympäristössä, on geneettisellä algoritmilla vaivalloista ja saavutettu tulos pätee vain ratkaisun ympäristöön. Dynaamisen ongelman ratkaisuun voisivat toimia paremmin esimerkiksi neuroverkot.

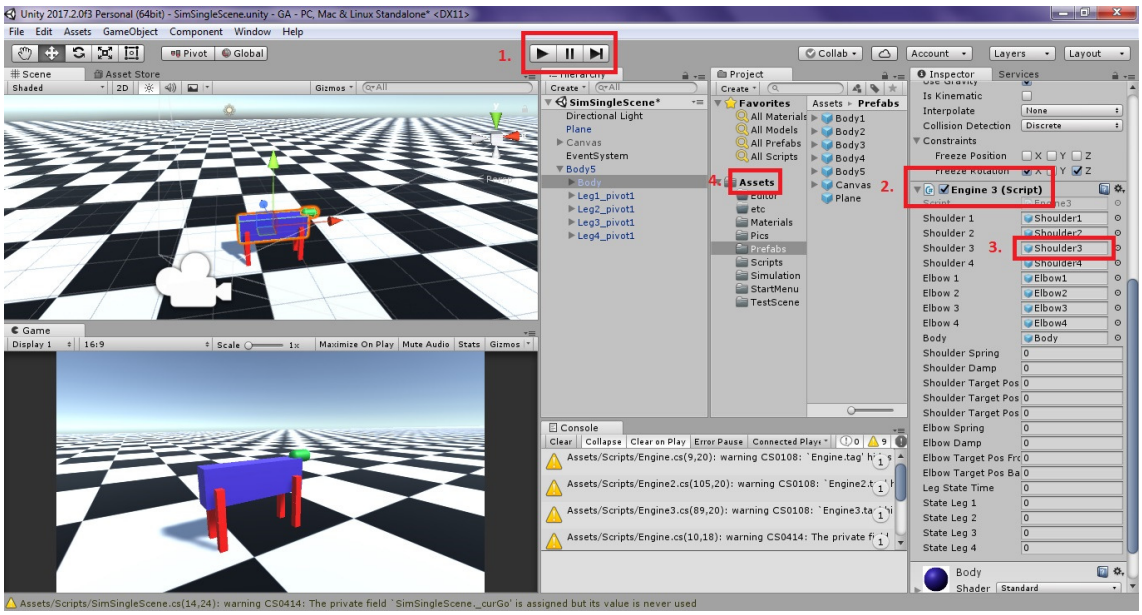
Geneettinen algoritmi ei välttämättä löydä globaalisti optimia ratkaisua ongelmaan, koska sen toimintaan vaikuttaa pitkälti se, kuinka monimuotoinen sen alkupopulaatio on. Tästä syystä tulos voi jäädä paikalliseen optimiratkaisuun, joka ei välttämättä ole paras, mutta tarpeeksi hyvä. Paikalliseen optimiratkaisuun voidaan kuitenkin vaikuttaa esimerkiksi skaalautuvalla mutaatiolla. Skaalautuvassa mutaatiossa mutaation todennäköisyyttä kasvatetaan, kun ratkaisun kelpoisuus laskee tai on valmiiksi huono.

3 Unity 3D -pelimoottori

Insinöörityön toteutuksessa Unity oli keskeisessä roolissa, sillä Unitylla luotiin kaikki simulaation osat ja osien väliset yhteydet. Unity 3D on reaaliaikainen pelimoottori, jolla voi tehdä 3D- ja 2D-pelejä [8]. Tässä työssä keskityttiin Unityn 3D-puoleen.

Unityn mukana tulee ohjelmointia varten sen oma Monodevelop-ohjelmointiympäristö (Integrated development environment, IDE) tai vastaavanlainen, tässä työssä käytetty Visual Studio, pelilogiikan ohjelmointia varten. Kirjoitetun ohjelmakoodin toiminta voidaan nopeasti testata editorin pelitilassa (Play mode). Unity tukee uusissa versioissa C#-ohjelmointikieltä, kun vanhemmissa versioissa oli tuki myös Javascriptille.

Unityn ohjelmointimalli perustuu peliobjekteihin. Peliobjektiin kiinnitettyjä skriptejä tai vaikkapa sarananivel (hinge joint) -ominaisuutta kutsutaan Unityssa komponenteiksi (components). Pelilogiikkaa ohjelmoitaessa peliobjektiin kiinnitettyyn skripti-komponenttiin luodaan GameObject (peliobjekti) -muuttuja, johon voidaan editorissa raahata jokin toinen peliobjekti, kuten kuvasta 4 näkee. Näin objektin ominaisuuksiin ja mahdollisiin muihin skripteihin ja komponentteihin päästään käsiksi eri peliobjekteista. Nämä ominaisuudet kuuluvat kutakin luokkaa laajentavaan MonoBehaviour-luokkaan, jota ilman tällainen ohjelmointimalli ei olisi Unityssa mahdollinen.



Kuva 4. Unityn editor-näkymä, johon on merkitty tekstissä mainitut 1. pelitila (play mode), 2. yksi peliobjektin komponenteista, 3. skripti-komponentin peliobjektimuuttuja, johon on raahattu Shoulder3-peliobjekti, sekä 4. digitaaliset resurssit (assets), joihin lukeutuvat skriptit, materiaalit, kuvat, äänet ja monet muut resurssit.

Unityssa on erittäin hyvä dokumentointi, jossa neuvotaan yksityiskohtaisesti, kuinka eri asioita voidaan toteuttaa, ja lisäksi laaja ja aktiivinen yhteisö, jolta voi pyytää apua. Käyttäjän on helppo lisätä peliinsä ääniä ja muita digitaalisia resursseja (assets). Käyttäjä pystyy myös Unityssa animoimaan muissa ohjelmistoissa luotuja 3D-malleja, 2D-malleista puhumattakaan. Lisäksi editorissa on helppo luoda valikoilla, napeilla ja mittareilla käyttöliittymä (User interface, UI) peliin. Unityssa on myös fysiikkamoottori, jolla peliobjekteille voi lisätä esimerkiksi realistisen painovoiman, niveliä ja kehyksiä, jotka tarkkailevat törmäyksiä muiden peliobjektien kanssa.

Unitylla voi tuottaa sisältöä eri käyttöjärjestelmille ja useille eri konsoleille, kuten Windows, Linux, iOS sekä PS4 ja Xbox One. Pelien lisäksi Unitylla voi tuottaa animaatioita ja graafisesti realistisia renderöintejä esimerkiksi rakennusalan käyttöön. Unity mahdollistaa myös AR (Augmented reality, lisätty todellisuus)- ja VR (Virtual reality, virtuaalitodellisuus) -sisällön tuottamisen. Unity on ei-kaupalliseen ja opiskelijakäyttöön ilmainen ohjelmisto, jonka monipuolisuus mahdollistaa laaja-alaisen työskentelyn pelien ja monen muun alan parissa. [8.]

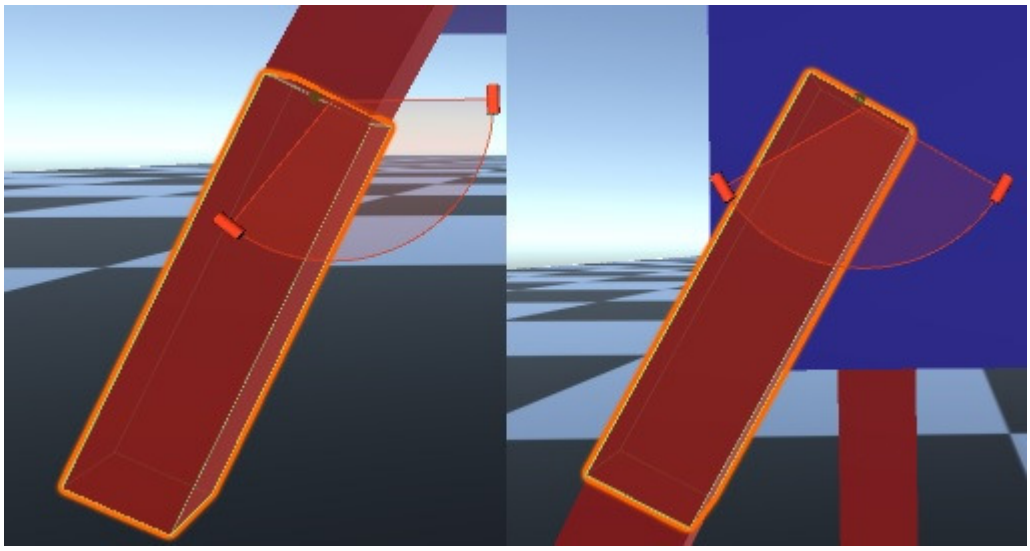
4 Geneettisen algoritmin ja simulaation toteutus

Tässä luvussa esitellään projekti pääpiirteissään, ja käydään läpi insinöörityön tekninen toteutus yksityiskohtaisesti kävelijän luonnista alkaen. Luvussa käsitellään myös kromosomin määritys, moottorin luonti, geneettisen algoritmin luonti sekä sen menetelmiin liittyviä valintoja, simulaation visualisointi, yhteenveto teknisen toteutuksen ohjelmallisesta rakenteesta sekä asiaa käyttöliittymästä.

4.1 Projektin kuvaus

Insinöörityön tarkoituksena oli opettaa geneettisen algoritmin avulla nelijalkainen kävelijä kävelemään tasaisella alustalla. Projektiaiheen valintaan vaikutti halu oppia ja ymmärtää koneoppimista, erityisesti evoluutioteoriaa soveltamalla. Kävelijän ja ympäristön luonti sekä kävelyn visualisointi toteutettiin Unitylla ja ohjelmointi C#-kielellä. Työ oli hyvä oppimiskokemus ja sitä voi käyttää pohjana samankaltaista projektia tehtäessä.

Projekti aloitettiin tutkimalla, miten nelijalkaisen kävelijän sai luotua Unitylla. Kävelijään tarvittiin keho, pää osoittamaan suuntaa sekä neljä raajaa, joiden kiertymiskulmaa rajoitettiin kuvan 5 mukaan halutun liikemallin saavuttamiseksi. Halutun liikemallin esikuvana olivat eläinmaailman nelijalkaiset eliöt.



Kuva 5. Kävelijän kyynärpään (vas.) ja olkapään (oik.) nivelten rajoitetut liikeradat.

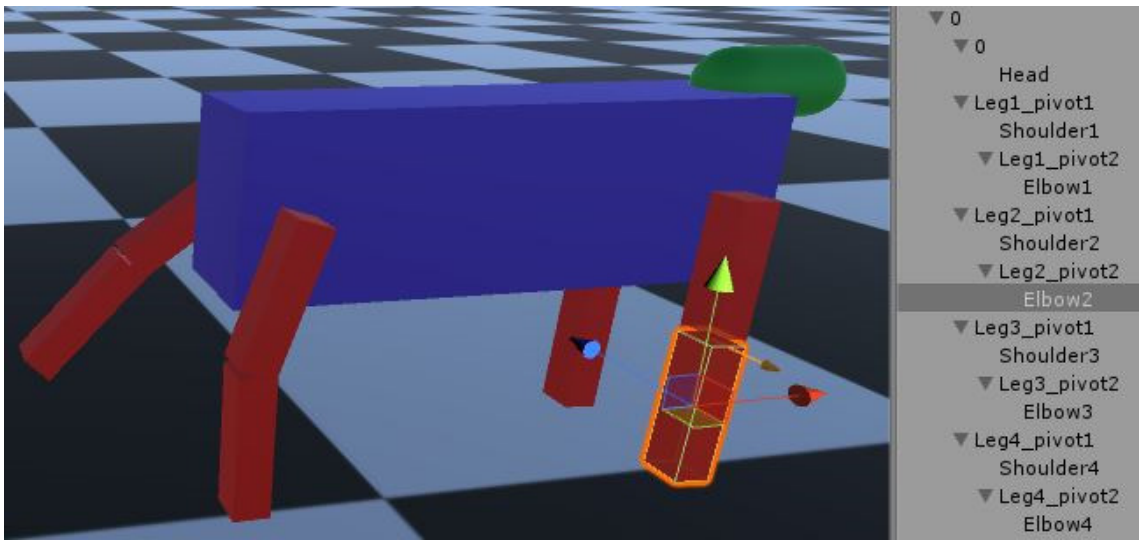
Kävelijän toteuttamisen jälkeen luotiin moottori liikuttamaan kävelijää. Erilaisia liikkumistapoja välitettiin moottorille kromosomeilla, jotka olivat optimointiongelman potentiaalisia ratkaisuja. Geneettinen algoritmi toteutettiin, mikä loi iteraatioita kromosomeista moottorin käytettäväksi.

Työn teknisen osan tavoitteiden täytyttyä parannettiin simulaation käyttäjäkokemusta metriikoita esittävällä kameralla sekä käyttöliittymällä, jonka avulla simulaation suorittaminen tuli käyttäjälle helpommaksi. Lisäksi käyttöliittymä mahdollisti koontiversion luonnin ohjelmasta.

4.2 Kävelijän rakenne

Unityssa mallinnettiin 3D-peliobjektilla kuutioista kävelijän keho ja neljä raajaa, joista kukin koostui kahdesta kuutiopeliobjektista. Lisäksi kehoon lisättiin pää kapseliobjektista osoittamaan kävelijän suuntaa. Raajojen osat liitettiin toisiinsa ja kehoon sarananivelillä (hinge joint), ja niille asetettiin kiertymisrajoitteet, joiden oli tarkoitus ohjata liikkumista eläinmaailmasta totuttuun suuntaan. Raajojen rakenteesta tehtiin yksinkertaisempi kuin eläinmaailman esikuvilla helpomman toteutuksen vuoksi. Kaikki kävelijän raajat ovat keskenään samanlaisia, vaikka todellisuudessa nelijalkaisten nisäkkäiden takajalat ovat rakenteeltaan hyvin samanlaisia verrattuna ihmisen alaraajaan. Kävelijän raajat muistuttavat kuitenkin ihmisen yläraajaa, mutta ilman kämmentä ja sormia.

Kokonaisuus koottiin tyhjän peliobjektin alle siten, että keho ja olkapäille luodut erilliset pivot-pisteet olivat hierarkiassa samalla tasolla. Olkapäiden pivot-pisteiden alla olivat olkapään kuutio-peliobjekti ja kyynärpään pivot-piste ja sen alla kyynärpään kuutio-peliobjekti, kuten kuvasta 6 voidaan nähdä. Tällä tavalla kokonaisuus oli helposti liikuteltavissa ja toisaalta eri peliobjektien geometria toimi oikein nivelten kanssa. Kävelijän kehon asetuksista asetettiin ”Is Trigger” päälle, minkä seurauksena jalat eivät törmänneet kehoon ja niiden liikkuvuutta saatiin parannettua.



Kuva 6. Kävelijän keho ja sen hierarkkinen toteutus. Raajan 2 kyynärpää on korostettuna oranssilla osoittamaan jalan eri osia.

Sarananivelten ja niiden avulla jalan osien liikutus esimerkiksi peliobjektiin kohdistuvilla rotaatiokomennoilla ei toiminut, sen sijaan nivelten omia ominaisuuksia jousi, vaimennus ja kohde (spring, damp, target) käytettiin saamaan jaloille kantavuutta. Näiden ominaisuuksien avulla kävelijä myös saatiin lopulta liikkumaan.

4.3 Kromosomin määritys

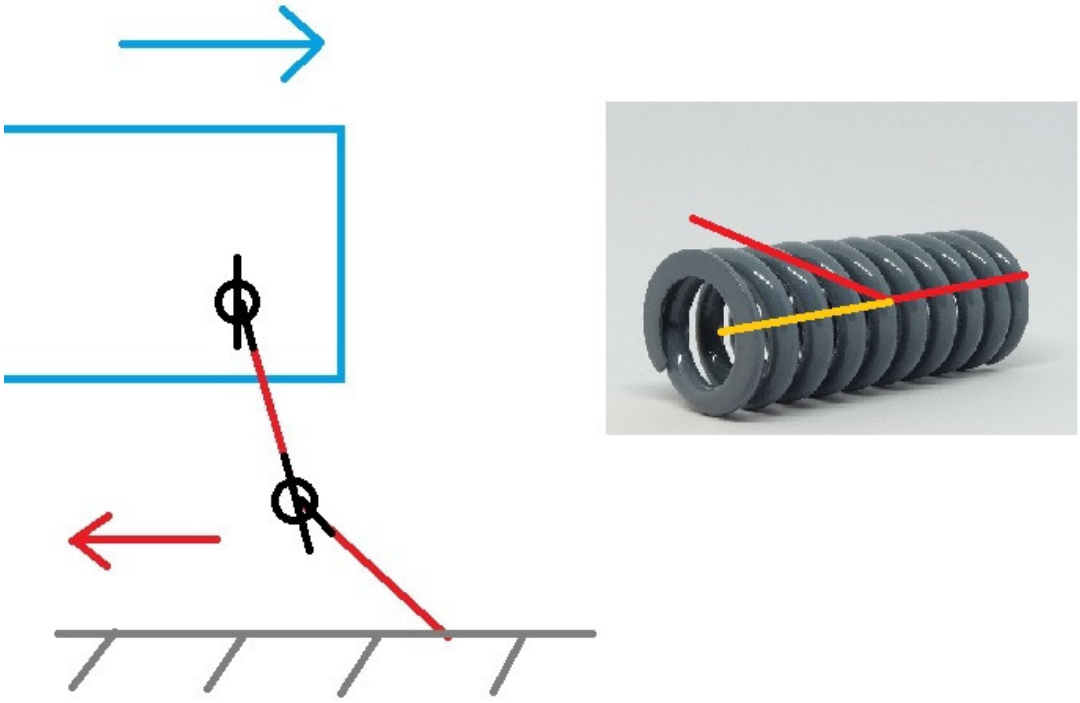
Kromosomi sisälsi listan ominaisuuksia, joita kehitettiin sukupolvesta toiseen ja jonka ansiosta kävelijän kävely lopulta kehittyi mahdollisimman hyväksi. Kromosomin sisältävän genomin ominaisuuksiksi päätettiin seuraavat 14 geeniä liukulukuina. Olkapäiden nivelten käsittelyyn määritettiin jousen voimakkuus, vaimennuksen voimakkuus sekä kohteet etu-, keski- ja taka-asennoille. Kyynärpäiden nivelten käsittelyyn määritettiin vastaavasti jousen ja vaimennuksen voimakkuudet sekä kohteet etu- ja taka-asennoille. Lisäksi kromosomissa oli myös yksittäisen jalan liikutukselle suoritusaika ja jokaiselle jalalle annettu järjestysluku, kuten esimerkkikoodista 1 voi nähdä.

```
shoulderSpring = walker._chromosome[0];
shoulderDamp = walker. _chromosome[1];
shoulderTargetPosFront = walker._chromosome[2];
shoulderTargetPosBack = walker._chromosome[3];
shoulderTargetPosMid = walker._chromosome[4];

elbowSpring = walker._chromosome[5];
```

```
elbowDamp = walker._chromosome[6];
elbowTargetPosFront = walker._chromosome[7];
elbowTargetPosBack = walker._chromosome[8];
legStateTime = walker._chromosome[9];

stateLeg1 = walker._chromosome[10];
stateLeg2 = walker._chromosome[11];
stateLeg3 = walker._chromosome[12];
stateLeg4 = walker._chromosome[13];
```

Esimerkkikoodi 1. Kävelijän moottorin arvoihin asetetaan kromosomin arvot.

Olka- ja kyynärpäiden kohteet määrittivät kävelijän raajojen liikeradan. Jousen voimakkuus vaikutti siihen, kuinka nopeasti ja voimakkaasti kohteiden vaihduttua olka- tai kyynärpää pyrki palautumaan stressittömään tilaan, jolloin jouseen ei kohdistunut taivuttavia voimia, jota on havainnollistettu kuvassa 7. Vaimennuksen voimakkuus sen sijaan vaikutti siihen, kuinka nopeasti jousen värähtely puolelta toiselle vaimeni stressittömään tilaan.

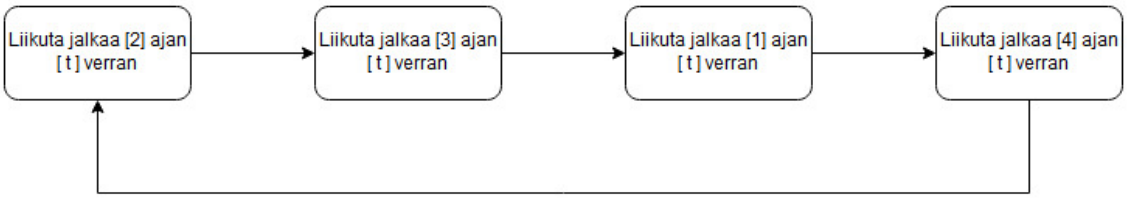


Kuva 7. Nivelten kohdeasennot ovat merkittyinä mustan ympyrän halkaisijoina, jolloin nivelten jouset ovat jännittämättömiä. Jouset ovat kuvan esimerkissä jännitetyssä tilassa ja pyrkivät näin kohteeseensa ja aiheuttavat kävelijälle eteenpäin vievän voiman. Oikealla on esimerkki jännittämättömästä jousesta ja siitä, kuinka jousta vasemmalla taivutetaan.

Kromosomin geenien liukulukuarvoille määritettiin jokaiselle omat minimi- ja maksimiarvot, joiden väliltä ne arvottiin satunnaisesti. Poikkeuksena geenien määrityksessä olivat indekseillä 10–13 olleet jalkojen järjestysnumerot. Järjestysnumeroita ei voinut arpoa satunnaisesti, koska muuten kävelijä saattoi liikuttaa esimerkiksi kaksi kertaa jalkaa 1 eikä kertaakaan jalkaa 3. Tästä syystä jalkojen järjestysluvut olivat valmiina listana, listan järjestys satunnaistettiin ja nämä arvot annettiin järjestyksessä jokaisen jalan muuttujaan.

4.4 Moottori

Engine2 (moottori) -luokka sai muuttujina kromosomin sisältämät liukulukuarvot ja käytti niitä kävelijän liikuttamiseen. Moottorissa oli kaksi metodia, MainMove ja MoveLeg, jotka hoitivat liikuttamisen FixedUpdate (päivitys) -metodissa, joka on Unityn sisäinen toistometodi. Metodit noudattivat tilakoneen (state machine) periaatteita, ja metodeista ensimmäinen hallinnoi jalkojen liikuttamista, kuten kuvasta 8 nähdään.

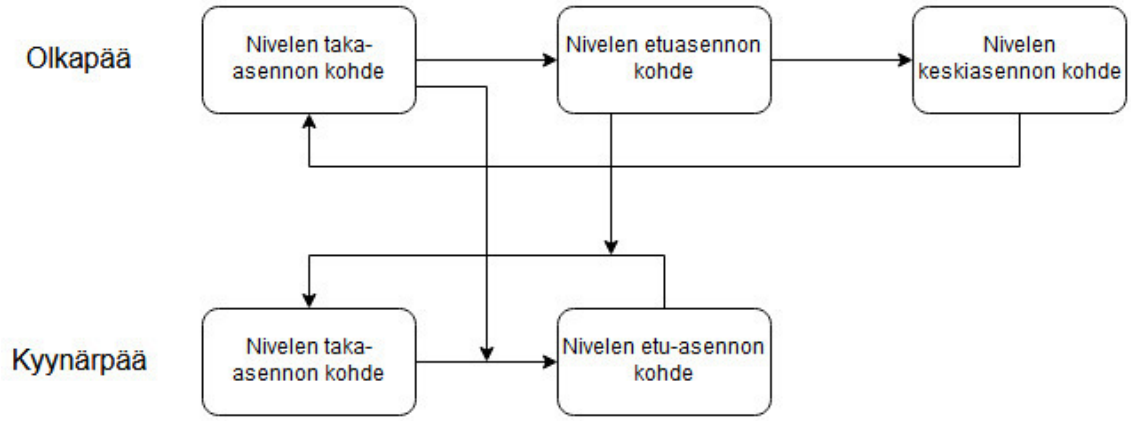


Kuva 8. Tilakone, joka antaa jokaiselle jalalle ajan [t] suoritusaikaa jalan liikuttamiseen jalan järjestysluvun mukaan. Tässä esimerkissä jalkoja liikutetaan järjestyksessä 2, 3, 1 ja 4.

Kävelijää liikutettiin tilakoneen kautta, jossa suoritusaika määritti, kuinka kauan kullekin jalalle annettiin aikaa liikkua ennen tilan vaihtumista seuraavalle jalalle. Tilan vaihtamiseen liittyivät myös jalkojen järjestysluvut, jotka määrittivät, missä järjestyksessä jaloille annettiin suoritusaikaa.

MoveLeg-metodi suoritettiin MainMove-metodin jokaisessa tilassa, ja se sai parametrina jalan numeron, jota sen tuli liikuttaa. Metodi valitsi toiminnan omassa tilakoneessaan saamallaan parametrilla ja muutti nivelen kohdearvoja tietyssä järjestyksessä, kuten kuva 9 osoittaa. Lähtötilanteessa jalat olivat kohtisuorat edettävää pintaa kohden ja ensimmäinen tila pyrki nostamaan jalan irti pinnasta kiertäen taka-asennon kautta. Seuraava tila pyrki asettamaan jalan eteenpäin koskematta pintaa ja lopulta suoristi sen. Viimeinen tila pyrki tuomaan jalan suorana lähtötilanteen asentoon, siten, että kohteen

muuttaminen aiheutti nivelen jouseen voiman, koska edettävä pinta pyrki estämään liikettä. Tämä sai jousen pyrkimään uuteen 0-arvoonsa kohteessaan, ja jalan raahaaminen pintaa pitkin aiheutti kävelijään eteenpäin vievän voiman.



Kuva 9. Olka- ja kyynärpään tilakoneet sekä olkapään tilakoneen vaikutus kyynärpään tilakoneeseen. Kyynärpää ei saa liikkua, ennen kuin olkapää on tietyssä asennossa.

Liikkeen kannalta oli tärkeää, että kyynärpään liike oli riippuvainen olkapään kiertymisasennoista, jolloin kokonaisliike oli tahdistettua. Unityn editorissa kävelijän kehon x- ja y-kiertymät rajoitettiin jalkojen oikeanlaisen toiminnan saavuttamiseksi. Tämä auttoi jonkin verran, mutta jalkojen liike tästä huolimatta loppui satunnaisesti, eikä tarkkaa syytä ongelmalle saatu selvitettyä. Kehon ”Is Trigger” -asetus auttoi tähän myös jonkin verran. Useimmiten ensimmäiseksi liikutettavan jalan, ja joskus jonkin muunkin jalan, toiminta rikkoontui.

4.5 Geneettisen algoritmin toteutus

Geneettiseen algoritmiin liittyvät asiat jaettiin kahteen luokkaan, joita olivat Chromosome (Kromosomi) ja Population (Populaatio). Chromosomen metodeja olivat konstruktori (Chromosome), CalcFitness, Crossover, Mutate ja RandGene. Populationin metodeja olivat puolestaan konstruktori (Population), NewPopulation, GetParent ja CalcFitness.

Aluksi Manager-luokassa luotiin uusi ilmentymä Population-luokasta luokan konstruktoria kutsumalla. Konstruktorilla välitettiin luokan ilmentymän jäsenmuuttujille lähtötietoja GA:n tulevia iteraatioita varten. Näitä olivat populaation koko, kromosomin koko,

satunnaismuuttuja, elitismillä valittavien yksilöiden määrä ja mutaation todennäköisyys. Jäsenmuuttujien alustamisen jälkeen konstruktori kutsui toistorakenteessa Chromosome-luokan konstruktoria. Tämä konstruktori välitti Chromosome-ilmentymän koon, eli kromosomin pituuden sekä satunnaismuuttujan ja totuusarvon, joka mahdollisti ilmentymän luonnin. Tämän jälkeen konstruktori loi listan satunnaisista geeneistä RandGenella. RandGene-metodi palautti liukulukuja perustuen kromosomin indeksin arvoon, eli jokaisella indeksillä oli ennalta määritetty liukulukujen alue, jolta luku satunnaisesti arvottiin.

Poikkeuksena satunnaisuuteen olivat kromosomin arvot indeksinumeroilla 10–13, jotka olivat jalkojen suoritusta varten annetut järjestysluvut. Nämä luvut eivät voineet olla täysin satunnaisia, koska muuten liikuttelujärjestyksessä saattoi olla esimerkiksi kaksi kappaletta jalkoja 1 ja nolla kappaletta jalkoja 3. Nämä käsiteltiin siksi yhtenä kokonaisuutena siten, että järjestysluvut olivat valmiissa listassa, listan järjestystä muutettiin satunnaisesti ja jokaiselle jalalle oli tasan yksi järjestysnumero.

Kun Chromosome-luokka oli palauttanut Population-luokalle vaaditun määrän yksilöitä, oli ensimmäinen populaatio valmis testattavaksi Unityssa. Silloin Manager loi yhtä suuren määrän kävelijöitä ja kävelypintoja kuin populaation koko oli. Manager loi pinnat CreatePlanes- ja kävelijät CreateWalkers-metodeissa. CreateWalkers etsi jokaisen kävelijän sisältämät peliobjektit ja asetti kromosomin arvot peliobjektien Engine2:een InitGAValues-metodin avulla.

Unityn simulaatiossa mitattiin kävelijöiden kulkema matka ja siihen käytetty aika. Kävelijöiden edistymistä seurattiin, ja vain aktiivisesti edistyvien yksilöiden simulaatiota jatkettiin. Kävelijän oli edettävä 5 sekunnin aikana 0,2 yksikköä x-suunnassa, jotta yksilön simulaatiota jatkettiin. Kävelijän keräämä tieto edetystä matkasta ja siihen käytetystä ajasta lähetettiin kävelijää vastaavan populaation yksilön ilmentymään odottamaan seuraavaa vaihetta.

Populaation kaikkien yksilöiden simulaatioiden loputtua Manager kutsui populaation ilmentymän NewPopulation-metodia. NewPopulationin ensimmäinen tehtävä oli laskea kaikkien edellisen populaation yksilöiden kelpoisuudet. NewPopulation kutsui omaa CalcFitness-metodiaan, joka puolestaan kutsui Chromosome-luokan CalcFitness-metodia. Tämä metodi laski yksilön kelpoisuuden painotetulla summalla, jossa edetyn matkan painoarvo oli 0,85 ja nopeuden painoarvo 0,15, kuten esimerkkikoodissa 2 voi nähdä.

0,85:n verran kelpoisuutta kävelijä ansaitsi 50 yksikön etenemällä ja 0,15:n verran kelpoisuutta 0,3:n nopeudella. Metodi palautti arvon Populationin CalcFitnessille, joka otti muistiin populaation parhaan yksilön kelpoisuuden sekä päivitti koko simulaation siihen mennessä parhaan kelpoisuuden muistiin.

```
public float CalcFitness()
{
      float dist = 0f;
      float speed = 0f;

      dist = _endPos - _startPos;
      if (dist > 50f)
            dist = 50f;
      speed = dist / (float)_runTimeSecs;
      if (speed > 0.3f)
            speed = 0.3f;

      _fitness = (float)(dist / 50f * 0.85f + speed / 0.3f * 0.15f);

      return _fitness;
   }
```

Esimerkkikoodi 2. Metodi laskee yksilön edetyn matkan ja nopeuden perusteella kelpoisuuden painotetusti. Matkalle ja nopeudelle on asetettu maksimiarvot 50 ja 2, jotka saavutettuaan yksilön kelpoisuus on 1.

Seuraavaksi yksilöt järjesteltiin kelpoisuuksien mukaan, siten, että parhaan kelpoisuuden omaava yksilö oli ensimmäisenä. Järjestelyn jälkeen alettiin luoda uutta sukupolvea, aloittaen elitismistä. Elitismissä valittiin 5 edellisen populaation parhaiten pisteytettyä yksilöä ja siirrettiin ne sellaisenaan uuteen populaatioon. Tällä tavoin uusi populaatio pärjäsi vähintään yhtä hyvin kuin edellinen. Elitismillä valittujen yksilöiden pärjääminen uudessa sukupolvessa oli epävarmaa, kun kävelijän kehon asetus ”Is Trigger” ei ollut päällä. Silloin edellisessä populaatiossa hyvin pärjänneen yksilön jalat saattoivat jumiutua uudessa sukupolvessa. ”Is Trigger” paransi huomattavasti GA:n toimintaa ja oli siksi yksi tärkeimmistä yksittäisitä ominaisuuksista projektin teknisessä toteutuksessa.

Elitismin jälkeen loput uuden populaation paikat täytettiin risteyttämällä edellisen populaation yksilöitä keskenään. Kutsuttiin esimerkkikoodissa 3 nähtävää GetParent-metodia kaksi kertaa, joissa kummassakin arvottiin uuden yksilön vanhempi osittain satunnaisesti turnajaisvalinnalla. Turnajaisvalinnassa valittiin ensin satunnaisesti populaatiosta kolme yksilöä ja näistä yksilöistä valittiin korkeimman kelpoisuuden omaava yksilö vanhemmaksi. Tällä tavalla painotettiin parempien yksilöiden valintaa, ja samalla varmistettiin populaation monimuotoisuuden säilymistä, sillä näin valittu paikallinen maksimi ei välttämättä ollut koko populaation tasolla korkea kelpoisuus.

```
Chromosome GetParent()
{
      Chromosome[] parentCandidates = new Chromosome[3];
      for (int i = 0; i < 3; i++)
      {
            int randNumber = _random.Next(_populationSize);
            parentCandidates[i] = _population[randNumber];
      }
      int index = 0;
      float max = parentCandidates[0]._fitness;
      for(int j = 0; j < parentCandidates.Length; ++j)
      {
            if(parentCandidates[j]._fitness > max)
            {
                  index = j;
            }
      }
      return parentCandidates[index];
}
```

Esimerkkikoodi 3. Metodi etsii satunnaisesti valittujen kolmen yksilön joukosta parhaan kelpoisuuden omaavan yksilön ja palauttaa sen.

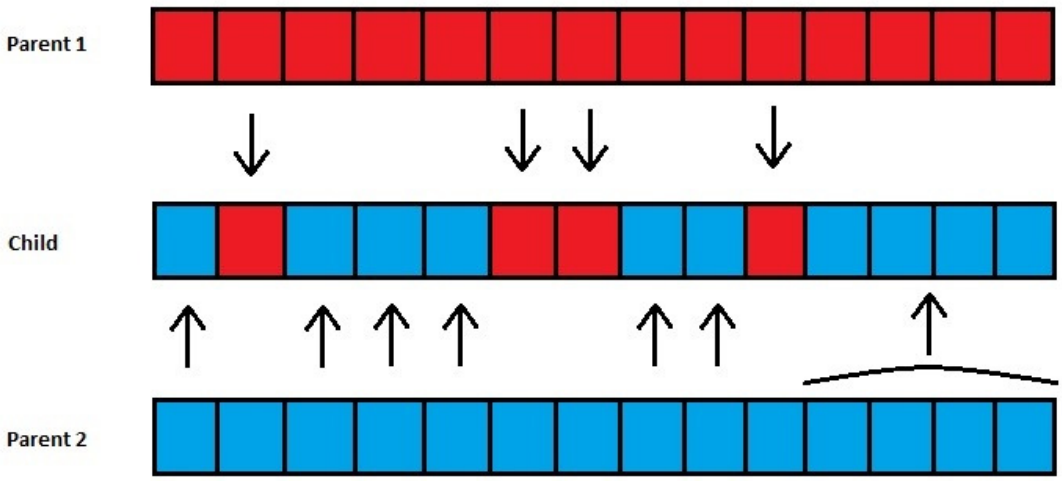
Kun vanhemmat oli valittu, lapsen (child) genomin geenit arvottiin satunnaisesti vanhempien geeneistä Chromosome-luokan Crossover-metodissa. Kummankin vanhemman geeneillä oli joka geenin kohdalla yhtäläiset mahdollisuudet tulla valituksi esimerkkikoodin 4 ja kuvan 10 esimerkkien mukaisesti.

```
public Chromosome Crossover(Chromosome otherParent)
{
      Chromosome child = new Chromosome(_chromosome.Length, _random,
      initPopulation: false);

      for (int i = 0; i < _chromosome.Length; i++)
      {
            if (i < 10)
            {
                  child._chromosome[i] = _random.NextDouble() < 0.5 ?
                  _chromosome[i] : otherParent._chromosome[i];
            }
            else
            {
                  if (_random.NextDouble() < 0.5)
                  {
                        child._chromosome[10] = _chromosome[10];
                        child._chromosome[11] = _chromosome[11];
                        child._chromosome[12] = _chromosome[12];
                        child._chromosome[13] = _chromosome[13];
                  }
                  else
                  {
                        child._chromosome[10] = otherParent._chromosome[10];
                        child._chromosome[11] = otherParent._chromosome[11];
                        child._chromosome[12] = otherParent._chromosome[12];
                        child._chromosome[13] = otherParent._chromosome[13];
                  }
            break;
            }
      }
```

```
        return child;
}
```

Esimerkkikoodi 4. Lapsikromosomin muodostus arpomalla satunnaisesti vanhempien geeneistä. Viimeiselle neljälle geenille on käsittely erikseen niiden toisistaan riippuvuuden vuoksi.

Kuten alkupopulaatiota luodessa tuli risteytyksessäkin käsitellä neljää viimeistä geeniä yhtenä kokonaisuutena niiden keskinäisen riippuvuuden takia, ettei jalkojen järjestyslukuihin tullut epäjohdonmukaisuuksia.



Kuva 10. Lapsen (child) jokainen geeni on satunnaisesti arvottu jommankumman vanhemman (parent) vastaavasta geenistä. Poikkeuksena viimeiset neljä geeniä, jotka ovat toisistaan riippuvaisia ja ne täytyy tuoda kaikki samalta vanhemmalta.

Kun lapsen genomi oli valmis, se altistettiin vielä mutaatiolle Chromosome-luokan Mutate-metodissa. Mutate sai parametrinaan hyvin pienen liukuluvun, noin 0,05–0,10, mikä merkitsee 5–10 %:n todennäköisyyttä mutaatiolle. Mutatessa jokainen geeni käytiin läpi, ja jos arvottu satunnaisluku oli pienempi kuin mutaation todennäköisyys, geeni korvattiin uudella satunnaisella geenillä. Risteytyksen ja alkupopulaation tapaan tässäkin neljää viimeistä geeniä käsiteltiin omana kokonaisuutenaan, kuten esimerkkikoodista 5 havaitaan. Mutateen lisättiin ominaisuus, joka skaalasi asetettua mutaation todennäköisyyttä 0,1:n verran, eli 10 % suuremmaksi, jos edellisen sukupolven paras kelpoisuus oli alle 0,08. Tällä haluttiin varmistaa, ettei GA ajautunut umpikujaan simulaation alussa, jos populaatio sisälsi liian huonoja yksilöitä.

```
public void Mutate(float mutationRate, Chromosome best)
{
      if(best._fitness < 0.08f)
      {
            mutationRate += 0.1f;
      }
      for(int i = 0; i <_chromosome.Length; i++)
      {
            if(i < 11 && _random.NextDouble() < mutationRate)
            {
                  _chromosome[i] = RandGene(i);
                  if(i == 10)
                  {
                        _chromosome[11] = _legs[1];
                        _chromosome[12] = _legs[2];
                        _chromosome[13] = _legs[3];
                        break;
                  }
            }
      }
}
```

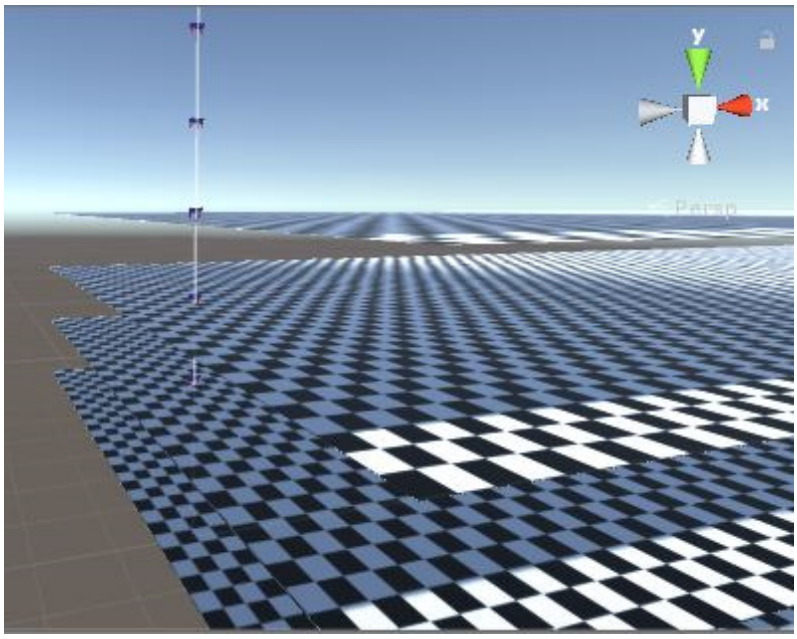
Esimerkkikoodi 5. Metodissa genomin geenit altistetaan yksitellen mutaatiolle, eli geeni saatetaan korvata uudella satunnaisluvulla. Viimeiset neljä geeniä on tässä tapauksessa käsiteltävä omana kokonaisuutenaan.

Mutaatiolle altistuksen jälkeen lapsi lisättiin populaatioon ja uusia lapsia tuotettiin niin kauan, kunnes populaatio oli täynnä. Lopuksi kaikki populaation yksilöt jälleen testattiin Unityn simulaatiossa ja kelpoisuuden selvityksen uusi iteraatio alkoi ja jatkui, kunnes ennalta määrätty sukupolvien määrä tuli täyteen tai tarpeeksi hyvä yksilö löytyi. Optimaalisessa ratkaisussa yksilön kelpoisuus on 1,0. Käytännössä simulaatio voitiin lopettaa hieman pienemmälläkin kelpoisuudella riippuen siitä, millainen ratkaisu riitti simulaation lopputulemaksi. Tässä kokoonpanossa kelpoisuuden alarajaksi asetettiin 0,95, joka varmasti riitti tarpeeksi hyvän ratkaisun saamiseksi ja lyhensi simulaatioaikaa.

Simulaation parhaan kelpoisuuden omaavien yksilöiden kelpoisuus ja genomin arvot tallennettiin tekstitiedostoon myöhempää tarkastelua varten. Näin käyttäjän ei tarvinnut olla seuraamassa simulaatiota koko ajan, sillä simulaatio saattoi venyä tuntien mittaiseksi. Lisäksi simulaation päättänyt paras yksilö ei välttämättä ollut sellainen, jota haettiin. Esimerkiksi hyvin nopeasti loppuun edennyt yksilö, jonka kaksi jalkaa eivät liikkuneet, vaan tasoittivat liikkumista, tuotti kelpoisuuden 1,0 olematta hyväksyttävä ratkaisu. Tämän vuoksi tiedostoon tallennettiin myös huonomman kelpoisuuden omaavia yksilöitä, joiden liikkuminen saattoi olla hyväksyttävää, alkaen yksilöistä, jotka etenivät koko 50 yksikön matkan.

4.6 Simulaation visualisointi

Simulaation alussa luotiin 50 fyysisesti samanlaisia kävelijää ja pintaa, joilla kävelijöiden oli tarkoitus kävellä, kuten kuvassa 11 on havainnollistettu. CameraScript-luokka piti kirjaa aktiivisten kävelijöiden body (keho) -peliobjekteista. Kun kävelijät etenivät pintaa pitkin koordinaatiston x-suunnassa, välittyi CameraScript-luokalle tieto niiden etenemisestä. Etenemien avulla CamTarget-metodi kävi läpi kaikki aktiiviset yksilöt ja vertasi niiden etenemiä ja siirsi kameran kuvaamaan sillä hetkellä eniten edennyttä yksilöä. Mikäli tämä yksilö ei edennyt tarpeeksi nopeasti ja sen simulaatio lopetettiin, siirtyi kamera uuteen, sillä hetkellä eniten edenneeseen yksilöön, kunnes kaikkien yksilöiden simulaatiot päättyivät.



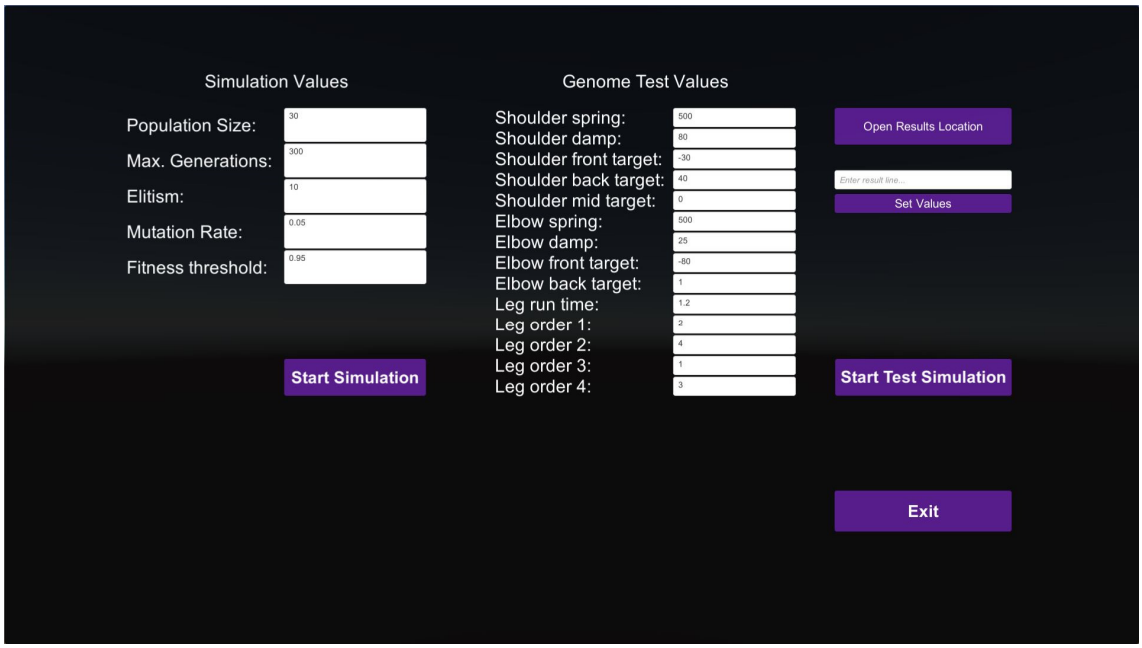
Kuva 11. Manager-luokka luo kävelijöitä ja kävelytasoja yhtä paljon, kuin on populaation koko. Iteraatiot luodaan päällekkäin.

Kameran yläriviin lisättiin paneeleita, jotka havainnollistivat käyttäjälle simulaation tilaa päivittämällä metriikoita. Paneeleiden kautta välitettiin senhetkisen sukupolven numero ja tieto sukupolvien maksimimäärästä. Toisessa paneelissa näytettiin sillä hetkellä kuvattavan kävelijän tunnusnumero ja aktiivisten kävelijöiden määrä senhetkisessä sukupolvessa. Kolmannessa paneelissa oli edellisen sukupolven paras kelpoisuus ja neljännessä koko simulaation paras kelpoisuus. Viidennessä paneelissa näytettiin yksilön kulkema matka ja maksimimatka, joka ilman nopeudesta saatavaa lisäkelpoisuutta nosti

kelpoisuuden korkeintaan 0,85:een. Viimeisessä paneelissa ilmoitettiin koko simulaatioon kulutettu aika.

4.7 Käyttöliittymä

Ohjelmaan toteutettiin alkuvalikko, jonka kautta käyttäjä pystyi asettamaan suoritettavan simulaation raja-arvoja. Kenttiin asetettiin oletusarvot, ettei kaikkia arvoja tarvinnut kirjoittaa joka kerta uudestaan ja myös ohjeistavaksi lukujen suuruusluokasta, kuten kuvasta 12 voi nähdä.



Kuva 12. Vasemmalla suoritettavan simulaation tekstikentät oletusarvoineen sekä simulaation aloitusnappi. Oikealla testikävelijän genomiarvot, nappi simuloitujen tulosten tietueeseen, syötekenttä tulosriville ja nappi sen jakamiseksi geenikenttiin sekä testikävelijän simulaation aloitusnappi.

Kun simulaatiosta oli saatu hyväksyttäviä ratkaisuja, ne tallentuivat ohjelmatiedostojen Results (tulokset) -kansioon. Näihin tekstitiedostoihin pääsi helposti käsiksi Open Results Location -napin kautta, josta simulaation tulokset kopioitiin rivi kerrallaan sille varattuun tekstikenttään, josta ne puolestaan siirtyivät niille varattuihin geenien tekstikenttiin Set Values -napin painalluksella. Tiedostossa rivin ensimmäinen arvo oli kelpoisuus, eikä sitä käytetty tekstikenttiin. Kun halutut arvot oli asetettu, genomin toimintaa pystyi testaamaan yksittäisellä kävelijällä. Testiympäristö auttoi käyttäjää ratkaisun valinnassa,

koska parhaan kelpoisuuden omaava ratkaisu ei välttämättä ollut paras ratkaisu. Kummankin simulaation pystyi lopettamaan Esc-painikkeen kautta ilmestyvästä valikosta. Ratkaisuja etsivän simulaation loputtua näytölle tuli ilmoitus loppumisesta ja käyttäjä pääsi palaamaan alkuvalikkoon.

4.8 Luokkien väliset riippuvuudet

Taulukossa 1 käydään läpi projektissa käytettyjen skriptien luokat, niiden päätoiminnot sekä yhteydet toisiinsa, selkeämmän kuvan antamiseksi teknisen osuuden rakenteesta.

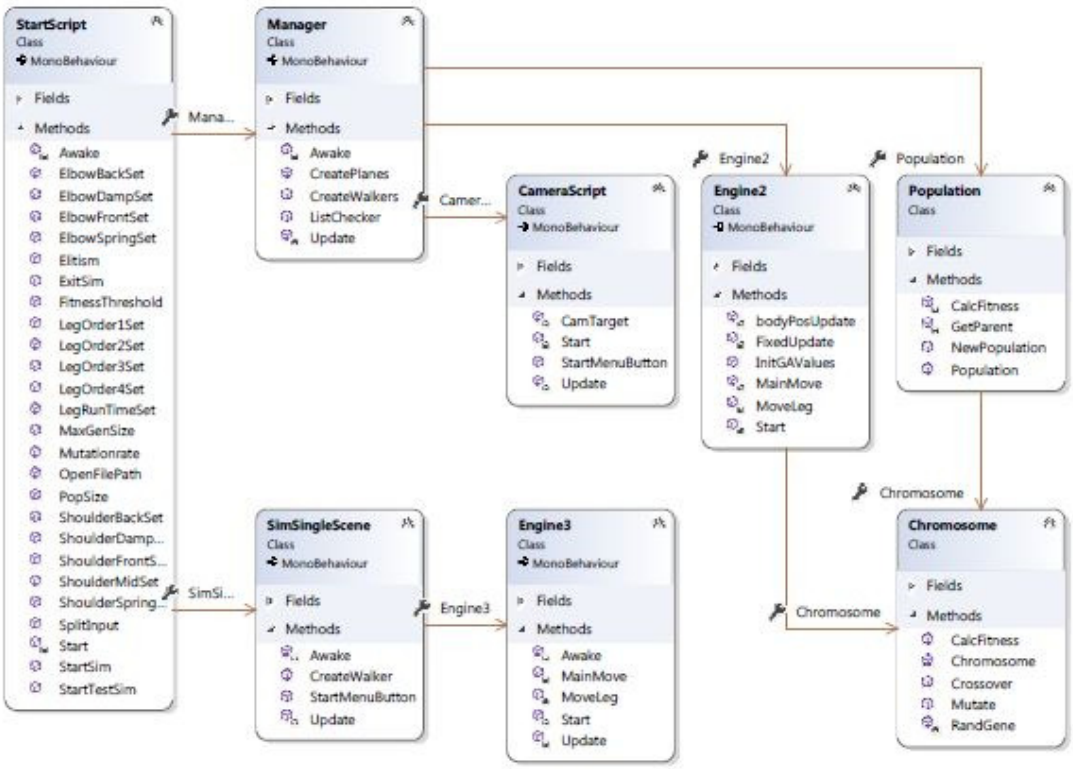
Taulukko 1. Projektissa käytettyjen skriptien luokat sekä luokkien toiminnan kuvaus.

Luokka	**Selite**
CameraScript	Kameran komponentti, ja sen tarkoitus on siirtää kameraa eniten edenneen yksilön mukana. Lisäksi tarkkailee simulaation UI-komentoja.
Chromosome	Luo yksittäiset kromosomit, laskee kelpoisuudet ja hoitaa uusien yksilöiden risteytyksen ja mutaation.
Engine2	Liikuttaa kävelijän raajan osia simulaatiossa ja välittää simulaation tulokset Chromosome-luokalle.
Engine3	Liikuttaa testisimulaation kävelijää käsin syötetyillä arvoilla.
Manager	Vastuussa simulaation kävelijöiden ja pintojen luonnista, lähtöarvojen välityksestä GA:lle, simulaation lopetuskriteerien tarkkailusta sekä tulosten kirjoittamisesta tiedostoon.
Population	Luo alkupopulaation Chromosome-luokan avulla ja luo uuden populaation valitsemalla uusien yksilöiden vanhemmat ja risteyttää ne Chromosomen metodeilla.
SimSingleScene	Luo testisimulaation kävelijän ja tarkkailee testisimulaation UI-komentoja.
StartScript	Käsittelee käyttäjän syötteitä alkuvalikossa ja välittää ne simulaatiolle tai testisimulaatiolle.

Ohjelman käynnistyessä StartScript alustettiin alkuvalikossa. StartScriptissä käyttäjän syötteet välitettiin joko simulaatioon tai testisimulaatioon, riippuen siitä, kumpaa haluttiin suorittaa. Simulaatiota aloitettaessa alustettiin Manager, joka loi vaaditun määrän kävelijöitä ja ilmentymän Populationista. Kävelijät puolestaan alustivat sisältämänsä Engine2:n, jonka muuttujiin tulivat arvot Populationin ilmentymään luotujen Chromosomen

ilmentymien kautta. CameraScript puolestaan tarkkaili näitä Managerin luomia kävelijöitä ja seurasi kohteita kameralla.

Testisimulaatiota suoritettaessa SimSingleScene alustettiin ja luokka loi yhden kävelijän ja tämä kävelijä puolestaan alusti sisältämänsä Engine3:n. Nämä riippuvuudet ovat nähtävissä kuvasta 13. SimSingleScene toimi Managerin lailla oman simulaationsa hallinnoivana osana.



Kuva 13. UML-kaavio luokkien välisistä riippuvuuksista.

Population ja Chromosome olivat skriptejä, joita ei kiinnitetty komponentteina peliobjekteihin, vaan ne toimivat taustalla staattisena. Kaikki muut skriptit sen sijaan perivät MonoBehaviour-luokan, minkä seurauksena ne täytyi kiinnittää komponenttina esimerkiksi tyhjään peliobjektiin. MonoBehaviour vaadittiin, kun käytettiin Unityn oman ohjelmointimallin ominaisuuksia.

4.9 Projektin toteutuksen haasteet

Kävelijä yritettiin aluksi luoda hierarkiaksi, jossa keho oli hierarkian juuri, olkapäät sen alaisina ja kyynärpäät olkapäiden alaisina. Sarananivelten kanssa tällainen hierarkia kuitenkin aiheutti peliobjekteihin geometrisia muodonmuutoksia ja hierarkia täytyi rakentaa uudelleen ja käyttää tyhjiä peliobjekteja niin hierarkian juurena kuin jokaisen jalan osan vanhempana. Lisäksi tyhjät peliobjektit nivelten kohdalla auttoivat kävelijän geometrian muokkauksessa siten, ettei kokonaisuuden geometria hajoa yhtä osaa muokattaessa.

Kävelijän liike koetettiin saada aikaiseksi yksinkertaisesti kiertämällä esimerkiksi kyynärpäätä sen vanhemman (parent) tyhjä-peliobjektin suhteen. Tämä osoittautui kuitenkin mahdottomaksi, sillä ilmeisesti kierto ilman fysiikkaa ei sisällä voimaa vastustaa painovoimaa ja kävelijä lyyhistyi heti simulaation alussa. Lopulta löytyi internetistä hyvin suppeasti kuvattu ratkaisu, jossa nivelten ominaisuuksia, kuten jousi, vaimennus ja kohde, oli käytetty vastaavanlaisen kävelijän liikkeelle saamiseksi [9]. Tätä ratkaisua käytettiin lopulta kävelijän liikkeen aikaansaamiseksi.

Engine-luokan toteutusta jalkojen liikkeelle testattiin siten, että kävelijää roikotettiin kehosta sen fysiikoiden ollessa kinemaattisia (is kinematic). Tällöin keho ja siihen liitetyt jalat eivät pudonneet maahan. Kun jalat oli viimein saatu toimimaan halutulla tavalla ja kävelijä päästettiin takaisin pinnalle, jalkojen toiminta häiriintyi selittämättömästä syystä, joka ei vielä ole yksiselitteisesti selvinnyt. Tämänhetkinen teoria on, että jalkoihin kohdistui vääntöä, joka jostain syystä rikkoi niiden toiminnan. Teorian perusteena on tämänhetkinen korjaus jalkojen toimintaan: siinä x- ja y-suuntien kiertoa kehossa on rajoitettu ja tämän seurauksena kävelijä alkoi kävellä ensin vaivattomasti, mutta välillä ongelmia silti esiintyi. Jalkojen toimintaa voi lisäksi häiritä simulaation ajaminen eri tietokoneilla, jolloin esimerkiksi kuvataajuus saattaa poiketa. Mahdolliset ohjelmointivirheet voivat myös osaltaan vaikuttaa jalkojen liikkeen toimimattomuuteen. Ongelmaan auttoi osaltaan kehon ”Is Trigger” -asetuksen käyttöönotto, jonka ansiosta jalat eivät väännön aiheuttaman liikkeen takia enää voineet osua kehoon ja jumiutua.

Käyttäjäkokemuksen parantamiseksi simulaatio oli tarkoitus rakentaa siten, että muutamia sukupolvia kuvattiin reaaliajassa, kehityksen havainnollistamiseksi käyttäjälle. Loput sukupolvien simulaatiot tuli suorittaa nopeutetusti, jotta simulaation kokonaisaika ei venyisi liian pitkäksi. Tätä ei voitu kuitenkaan toteuttaa täysin onnistuneesti, koska

simulaatiossa käytettiin Unityn fyysiikkamoottoria. Simulaation paikkansa pitävien tulosten saavuttamiseksi Engine-luokan MainMove-metodia tulee suorittaa FixedUpdate-metodissa Update-metodin sijaan. Updatessa aikaskaalan nopeutus toimi oikein, mutta fysiikan simulointi ei tapahtunut oikein. FixedUpdatessa sen sijaan fysiikat toimivat oikein, mutta seuraava päivitys tapahtui vasta, kun kaikki laskutoimitukset oli suoritettu loppuun. Tämä aiheutti simulaatiotilanteen visuaalisuuden jähmettymistä usein. Toisaalta välillä näytöllä näkyi sellaisen sukupolven reaaliaikaista simuloimista, mistä sitä ei pitänyt tapahtua ja esimerkiksi saman sukupolven viimeisen aktiivisen yksilön visualisointi tapahtui jälleen nopeutetusti. Nopeutus taas aiheutti sen, että yksilö saattoi edetä 0-ajassa, mikä aiheutti jälleen virheellisiä tuloksia.

Aikaa mitattiin DateTime.Now-komennolla, jonka lisäksi kokeiltiin Time.deltaTime- ja Time.fixedDeltaTime-komentoja, jotka kaikki aiheuttivat virheellisiä tuloksia nopeutetussa simulaatiossa. Tästä syystä nopeutuksesta luovuttiin tämän projektin osalta kokonaan.

5 Yhteenveto

Tulokset

Insinöörityöprojektin tarkoituksena oli perehtyä geneettisten algoritmien toimintaan ja ratkaista aiheeseen sopiva optimointiongelma niiden avulla. Optimointiongelmaksi valittiin kävelyn opetus nelijalkaiselle kävelijälle geneettisellä algoritmilla. Kävelijä ja sen ympäristö luotiin Unitylla ja ohjelmointi tehtiin C#:lla.

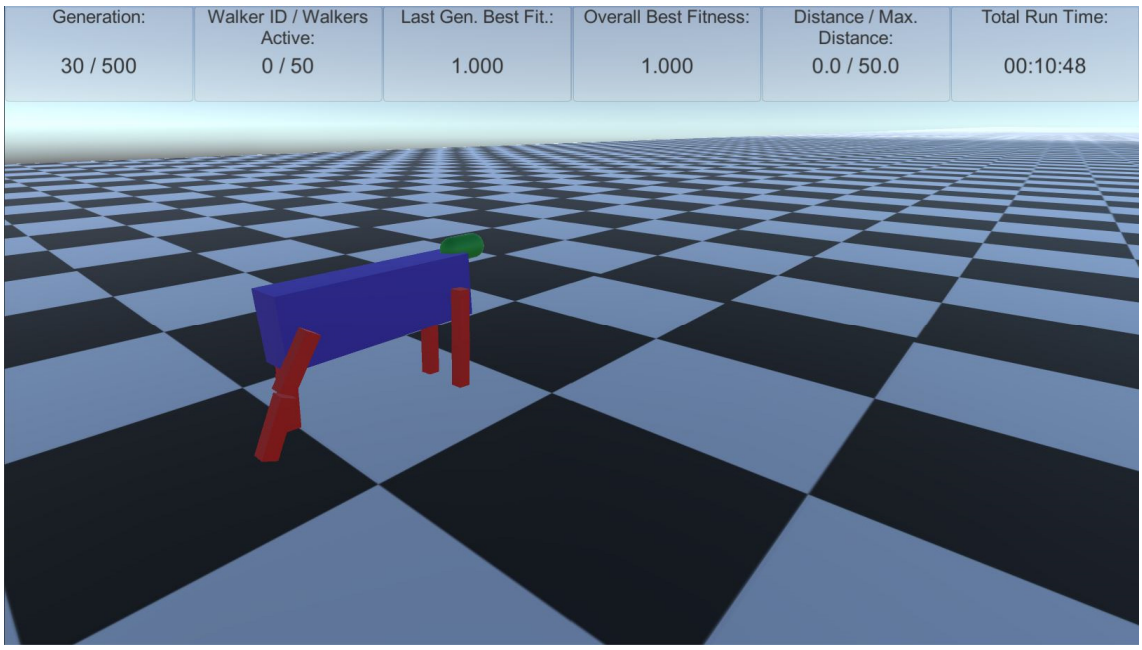
Kävelijä luotiin hierarkkisesti, ja se koostui kehosta, päästä ja neljästä jalasta, jotka kukin koostuivat olka- ja kyynärpäästä. Lisäksi hierarkiassa oli tyhjiä peliobjekteja jalkojen nivelten kohdalla sekä koko hierarkian juurena, jolla varmistettiin kävelijän liikuteltavuus ja peliobjektien geometrian muuttumattomuus toisiinsa nähden. Jalkojen osat kiinnitettiin toisiinsa sarananivelillä, minkä takia kävelijän liikuttaminen tapahtui nivelen ominaisuuksien jousi, vaimennus ja kohde avulla. Jousi määritti, kuinka voimakkaasti nivel pyrki pitämään rasittamattoman tilansa. Vaimennus sen sijaan määritti sen, kuinka nopeasti nivel asettui rasituksesta rasittamattomaan tilaan. Kohde taas määritti, missä nivelen

kiertymässä sijaitsi rasittamaton tila. Tähän tilaan jousi pyrki palautumaan, ja tämä aiheutti kävelijälle eteenpäin vievän liikkeen.

Geneettisen algoritmin toiminta saatiin ohjelmoitua, eikä sen toiminnassa ole löytynyt moittimista. GA:ta varten muodostettiin 14 liukulukuarvon kromosomi, jonka kelpoisuutta testattiin ja risteyttämällä kehitettiin paremmaksi. Kromosomin sisältämiä arvoja olivat olka- ja kyynärpäille jousen voimakkuus, vaimennuksen voimakkuus sekä kohteet etu- ja taka-asennoille ja lisäksi olkapäille kohde keskiasennolle. Näiden lisäksi kromosomissa oli suoritusaika jalan liikutukselle ja jokaiselle jalalle annettu järjestysluku.

Kävelijää liikuttamaan ohjelmoitiin moottori, jonka muuttujiin ja toimintaan vaikuttivat geneettisen algoritmin seurauksena kehittyvät kromosomin arvot. Kävelijän liikuttaminen perustui kahden sisäkkäisen tilakoneen toiminnalle. Tilakoneista ensimmäinen antoi joka jalalle kromosomissa määritetyn suoritusajan kyseisen jalan liikuttamiseen, minkä jälkeen tila vaihtui seuraavan jalan suoritukseen. Toinen tilakone puolestaan vaihtoi liikutettavan jalan olka- ja kyynärpään kohteita edellisen kohteen saavutettuaan siten, että kyynärpään tilan vaihtumiseen vaikutti myös olkapään kiertymisen sijainti.

Simulaation seuraamisen helpottamiseksi tehtiin sukupolven senhetkistä eniten edennyttä yksilöä seuraava kamera, jonka yläreunassa esitettiin simulaation tilan metriikkaa. Metriikoissa esitettiin senhetkisen sukupolven järjestysnumero sekä sukupolvien maksimimäärä, joka oli käyttäjän määritettävissä. Seurattavan yksilön tunnistenumero ja sukupolven aktiivisten kävelijöiden määrä sekä edellisen sukupolven ja simulaation senhetkisen parhaan ratkaisun kelpoisuudet näytettiin metriikoissa. Lisäksi metriikoissa esitettiin seuratun yksilön etenemä ja koko simulaatioon käytetty aika, kuten kuvasta 14 nähdään.



Kuva 14. Simulaatio on päättynyt, kun kelpoisuus 1 saavutettiin sukupolvessa 29. Kuvassa uusi populaatio on ehditty luoda, minkä takia metriikat eroavat tuloksesta.

Käyttäjää varten tehtiin myös alkuvalikko, jonka kautta simulaation raja-arvoja saatiin määritettyä helposti käyttäjän syötteillä. Valikosta käyttäjä pääsi myös testaamaan yksittäisiä ratkaisuja erillisessä yhden kävelijän ympäristössä, johon käyttäjä pystyi syöttämään omia arvojaan. Käyttäjän oli myös mahdollista hakea simulaation tekstitiedostoon tallentamia arvorivejä suoraan syötekenttään, joka tulkitsi ja jakoi rivin genomin geenisyötekenttiin.

Kävelyn opetus geneettisillä algoritmeilla saatiin onnistuneesti toteutettua. Simulaation onnistui tuottaa halutunlaisia eliömaailman inspiroimia liikemalleja, vaikka se tuotti lisäksi myös oudompia ratkaisuja. Toteutuksessa oli myös muutamia ongelmia sekä asioita, joita pystyisi jatkokehittämään.

Kehityskohdat

Toteutusta tehtäessä ilmeni seuraavanlaisia ongelmia, joista osa on kosmeettisia, osa mahdollisesti GA:ta parantavia ominaisuuksia ja osa GA:n toimintaa oletettavasti paljon heikentäviä ongelmia.

Kävelijän jalkojen toiminta rikkoontui välillä, mikä voi liittyä ainakin osin jalkojen kiertymiseen simuloinnin aikana. Kävelijän niveliin kohdistuu vääntäviä voimia simuloinnin aikana, ja vaikka nivelet on asetettu rikkoutumattomiksi, voi suunnittelematon vääntö kuitenkin rikkoa niiden toiminnan. Toisaalta osittainen aiheuttaja voi olla myös mahdolliset ohjelmointivirheet Engine2-luokassa tai jopa kuvataajuuden muutokset, erityisesti eri tietokoneilla suoritettaessa. Liikkumisen ongelmia tulisi joka tapauksessa tutkia lisää ja tarvittaessa kehitellä erilainen ratkaisu moottoriksi. Tästä syystä myös mahdolliset saadut hyvät ratkaisut eivät välttämättä pärjää enää seuraavassa sukupolvessa ja vaikeuttavat GA:n toimintaa, vaikka tähän auttoikin kävelijän kehon ”Is Trigger” -asetuksen aktivointi. ”Is Trigger” teki kehosta läpimentävän jaloille, eivätkä jalat niin helposti jumiutuneet sen vuoksi. Kävelijän ensimmäiseksi liikutettavassa jalassa ongelma ilmeni useimmin.

Kävelijän muokkausta varten hierarkiaa voisi vielä parantaa ja simulaation alkuvalikkoon lisätä valikon, jossa kävelijän geometriaa pystyisi helposti muokkaamaan. Esimerkiksi kävelijän jalanosien pituuksia ja kiinnityskohtia kehoon sekä kehon muotoa voisi muokata.

Geneettinen algoritmi ohjelmoitiin tätä projektia varten, eikä se siksi ole käytettävissä sellaisenaan muissa geneettistä algoritmia hyödyntävissä ongelmanratkaisuissa. GA:n ratkaisua voisi päivittää geneeriseksi, jotta sitä voisi käyttää muiden ongelmien ratkaisussa helpommin.

Projektissa keskityttiin ratkaisuun pääsemiseen eikä täydellisen ohjelmakoodin tuottamiseen. Tästä syystä koodia pystyy optimoimaan helpommin muokattavaksi ja luettavaksi.

Aikaskaalan toteuttaminen halutulla tavalla jätti toivomisen varaa. Simulaatiossa käytettiin Unityn fysiikkamoottoria, ja näin ollen oli käytettävä FixedUpdate-metodia, joka ei taipunut simulaation todelliseen nopeuttamiseen. Tähän voisi tutkia ratkaisua, kuinka pystytään toteuttamaan fysiikoihin perustuva simulaatio ja samalla nopeuttamaan simulaatiota useiden sukupolvien ajaksi tai saamaan selville, onko tämä edes mahdollista jollain tavalla. Simuloinnin nopeutus täytyi jättää pois toteutuksesta, mikä paransi tuloksia, mutta pidensi simulaatiota tuntien mittaiseksi.

Projektilla on useita kehityskohtia, joilla simuloinnin käyttäjäystävällisyyttä ja erityisesti geneettisen algoritmin toimintaa voisi parantaa, ja näitä suositellaan tutkittavan

vastaavanlaista projektia suoritettaessa. Tämä projekti antaa ohjeistavan suunnan vastaavanlaisille projekteille, ja projektia voi pitää onnistuneena muutamasta vakavasta ongelmasta huolimatta. Projekti oli myös erinomaisen hyvä ja mielenkiintoinen oppimiskokemus niin geneettisistä algoritmeista kuin Unityn käytöstäkin. Oli inspiroivaa nähdä evoluution menetelmät toiminnassa.

Lähteet

1 Heikkinen, Mikko. 2005. Evoluutio ja evoluutioteoria. Verkkoaineisto. <https://www.biomi.org/biologia/evoluutio/>. Luettu 8.10.2018.

2 Siitonen, Samuli. 2017. Hakuperustainen proseduraalinen pelien sisällön generointi geneettisellä algoritmilla. Kandidaatin työ. Lappeenrannan teknillinen yliopisto. Lutpub-tietokanta.

3 Aziz, Sami. 2007. Geneettinen Algoritmi. Opinnäytetyö. Satakunnan ammattikorkeakoulu. Theseus-tietokanta.

4 Genomi. 2016. Verkkoaineisto. Tieteen termipankki. <http://tieteentermipankki.fi/wiki/Nimitys%3Agenomi>. Luettu 22.10.2018.

5 Alleeli. 2016. Verkkoaineisto. Tieteen termipankki. <http://tieteentermipankki.fi/wiki/Biotekniikka:alleeli>. Luettu 22.10.2018.

6 Genetic Alghorithms – Parent Selection. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm>. Luettu 2.10.2018.

7 Genetic Alghorithms – Crossover. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm>. Luettu 2.10.2018.

8 Explore Unity. Verkkoaineisto. Unity Technologies. <https://unity3d.com/unity>. Luettu 19.10.2018.

9 Clueless about rigidbody based legs… . 2015. Verkkoaineisto. Unity Answers. <https://answers.unity.com/questions/895717/clueless-about-rigidbody-based-legs.html?childToView=901680>. Luettu 5.8.2018.