

Lauri Räsänen

Geneettisen algoritmin soveltaminen liikkumiseen Source-pelimoottorin Surf- pelimuodossa

Tradenomi
Tietojenkäsittely
Syksy 2020



KAMK • University
of Applied Sciences

Tiivistelmä

Tekijä: Räsänen Lauri

Työn nimi: Geneettisen algoritmin soveltaminen liikkumiseen Source-pelimoottorin Surf-pelimuodossa

Tutkintonimike: Tradenomi (AMK), Tietojenkäsittely

Asiasanat: geneettinen algoritmi, liikkuminen, pelit, Source-pelimoottori

Työn tarkoituksena oli tutkia geneettisen algoritmin sovellettavuutta monimutkaisen 3D-pelimaailman navigoinnissa hyödyntäen Source-pelimoottorin yleistämää Surf-pelimuotoa käytännön sovelluksena.

Työn teoriaosuus sisältää tietoa geneettisistä algoritmeista, niiden toimintaperiaatteesta, sekä sovellettavuudesta peleissä liikkumiseen. Työn käytännön osuudessa käydään tarkemmin läpi, kuinka ohjelmistosovellus on toteutettu, toteutuksen aikana ilmenneitä ongelmia ja ratkaisuja, sekä muita havaintoja.

Osana työtä kirjoitettu ohjelmistosovellus hyödyntää tavanomaista geneettistä algoritmia liikekomentojen tuottamiseen sekä SourceMod-ohjelmistorajapintaa vuorovaikutukseen Source-pelimoottorin kanssa. Sovelluksen lähdekoodi julkaistiin avoimena lähdekoodina hyödyntäen kolmatta versiota lisenssistä GNU General Public License.

Ohjelmistosovelluksen tuottamat tulokset ovat lupaavia ja toimivat todisteena sovelluksen toimivuudesta lyhyiden Surf-pelikenttien ratkaisemisessa, hyödyntäen samoja liikkumismekaniikoita kuin ihmispelaajat. Pidemmät pelikentät ovat vielä ongelmallisia algoritmin hakualueen eksponentiaalisen kasvun takia, mutta tämä ongelma on todennäköisesti ratkaistavissa hyödyntäen jatkokehitysosiossa esiteltyjä optimointimenetelmiä.

Abstract

Author: Räsänen Lauri

Title of the Publication: Applying a Genetic Algorithm to Movement in Source Engine's Surf Game Mode

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: genetic algorithm, movement, games, Source engine

The objective of this bachelor's thesis was to explore the applicability of genetic algorithms in navigating a complex 3-dimensional game environment. This was done by utilising the Surf game mode popularised by the Source engine as a practical example.

The theory section of this publication contains information on genetic algorithms, their operational principle, as well as, on their applicability in game movement. The practical part of this publication describes in greater detail how the application has been implemented, potential problems encountered during the implementation, solutions to these problems, and other observations.

The software written as a part of this thesis utilises a standard genetic algorithm for producing movement commands, and the SourceMod API for interfacing with the Source engine. The resulting software was open-sourced and released under the GNU General Public License version 3.

The results produced by the software are promising and demonstrate that a genetic algorithm can be used to solve short Surf stages, by using the same movement mechanics as real human players. Longer stages remain a challenge due to the exponential nature of the search space, but this problem may be solved by implementing the optimisation methods described in the further development section.

Sisällysluettelo

1	Johdanto	1
2	Geneettinen algoritmi	2
2.1	Käsite	2
2.2	Biologinen tausta.....	2
2.3	Algoritmin toimintavaiheet	2
3	Soveltaminen peleissä liikkumiseen	5
3.1	Ohjauskomennot.....	5
3.2	Mutaatio.....	6
3.3	Soveltuvuuden mittaaminen	7
3.4	Determinismi.....	8
3.5	Hyvät ja huonot puolet.....	8
4	Käytännön sovellus Source-pelimoottorin Surf-pelimuodossa	10
4.1	Source.....	10
4.2	Surf	11
4.3	Sovellettavuus	12
4.4	Ohjauskomennot.....	12
4.5	Hakualue.....	13
4.6	Soveltuvuuden mittaaminen	14
4.7	Tulokset	18
5	Jatkokehitys	22
5.1	Vaihtuva liikekomentojen toistamistaajuus	22
5.2	Tason paloittelu optimointia varten.....	22
6	Yhteenveto	25
	Lähteet	26
	Liitteet.....	28

Symboliluettelo

bitittäinen operaatio	bittipeitteelle tehtävä laskutoimitus, joka operoi yksittäisten bittien tasolla
bittipeite	yksittäisistä biteistä muodostuva sarja, jossa jokaisen bitin arvo kuvastaa erillisen muuttujan tilaa
determinismi	metafyysinen näkemys, jonka mukaan tapahtumien kulku on ennalta määrätty
geeni	geneettisen informaation yksikkö
hakualue	ongelman kaikkien mahdollisten ratkaisujen joukko
kromosomi	yksittäisistä geeneistä muodostuva sarja
laskenta-aika	algoritmin suorittamiseen vaadittu aika
modi	muokkaus, joka lisää uusia ominaisuuksia ohjelmistoon, lyhenne sanasta modifikaatio
mutaatio	yhdessä tai useammassa geenissä tapahtuva rakenteellinen muutos
populaatiogenetiikka	perinnöllisyystieteen haara, joka tutkii geneettisen tiedon muutosta
soveltuvuusfunktio	funktio, joka tiivistää ongelman ratkaisuehdotuksen soveltuvuuden yhdeksi lukuarvoksi
tekijäinvaihdunta	kromosomien katkeaminen ja niiden osien uudelleenjärjestäminen

1 Johdanto

Geneettiset algoritmit ovat julkisessa mediassa esiintyvistä hakualgoritmeista yksi suosituimmista. Geneettisiin algoritmeihin törmää usein tieteellisissä artikkeleissa, mutta myös jopa suosittu median kulttuurissa, kuten elokuvissa ja tv-sarjoissa. Geneettisillä algoritmeilla on pitkä tieteellinen historia lähtöisin 1950-luvulta ja niitä hyödynnetään usein käytännön optimointi- ja haakuongelmien ratkomiseen. Geneettisiä algoritmeja on sovellettu myös esimerkiksi virtuaalisen hahmon kävelemisen opettamiseen, mutta käytännön sovelluksia peleissä näkee harvemmin.

Työn tarkoituksena on tutkia geneettisten algoritmien soveltamista peleissä pelaajahahmon liikuttamiseen hyödyntäen samoja pelimekaniikoita kuin ihmispelaajat. Työn päämääränä on samalla selvittää, kuinka hyvin geneettisiä algoritmeja voi soveltaa liikkumiseen ja kartoittaa mahdollisia ongelmakohtia sekä muita huomioon otettavia asioita. Tämä työ syntyi oman mielenkiinnon johteesta geneettisiä algoritmeja ja pelejä kohtaan, ja sen ideana on myös toimia mielenkiinnon herättäjänä muille pelialalla toimiville henkilöille, mikä toivottavasti johtaa uusiin mielenkiintoisiin sovelluksiin peleissä.

Vaikka tämä työ keskittyy juuri pelaajahahmon liikuttamiseen, geneettisiä algoritmeja voi soveltaa myös muiden hahmojen tai objektien liikuttamiseen peleissä. Liikkeen tuottaminen ja optimointi peleissä algoritmeja hyödyntäen on ajankohtainen ja hyvin käytännön läheinen aihe pelialan keskuudessa.

Työ koostuu karkeasti kahdesta osasta. Ensimmäinen osuus sisältää teoriaa geneettisistä algoritmeista, niiden toimintaperiaatteesta sekä sovellettavuudesta peleissä liikkumiseen. Työn toisessa osuudessa käydään läpi käytännön sovelluksen toteutus, toteutuksen aikana ilmenneitä ongelmia ja ratkaisuja sekä muita havaintoja.

2 Geneettinen algoritmi

2.1 Käsite

Geneettinen algoritmi on erään tyyppinen evoluutioalgoritmi, jota usein käytetään optimointiongelmiin ratkomiseen. Evoluutioalgoritmit ovat evoluutioteoriaan ja biologiaan pohjautuvia hakualgoritmeja, joiden toiminta perustuu väestön evoluutioon toistettavien operaatioiden avulla. [1.]

2.2 Biologinen tausta

Geneettiset algoritmit hyödyntävät vahvasti evoluutioteoriassa esiintyvää luonnonvalintaa optimoidakseen populaatiota jotain tiettyä ongelmaa varten. Evoluutioteoriasta on lähes mahdoton kirjoittaa mainitsematta Charles Darwinin maailmankuulua julkaisua *Lajien synty* (engl. *The Origin of Species*), joka on toiminut vuosisatojen saatossa yhtenä tärkeimpänä kehitysopin perustajista. Geneettiset algoritmit hyödyntävät evoluutioteorian lisäksi populaatiogenetiikkaa geneettisen tiedon manipulointiin. Evoluutioteorian ja populaatiogenetiikan yhdistymä, synteettinen evoluutioteoria eli uusdarwinismi, toimii tärkeänä biologisena pohjana geneettisten algoritmien toiminnalle. [2.]

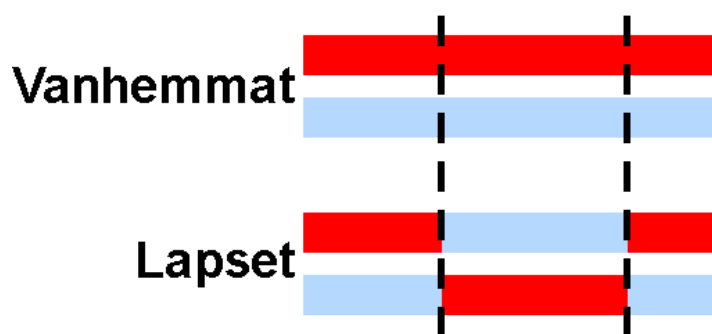
Geneettisen algoritmin populaation yksilöt, tai kromosomit, muodostuvat sarjasta geenejä eli geneettisen informaation yksiköistä. Näitä kromosomeja muokataan useampien sukupolvien aikana hyödyntäen erilaisia valintamenetelmiä, yksilöiden risteytystä ja mutaatiota. Geneettiset algoritmit lisäksi hyödyntävät luonnonvalinnassa esiintyvää elitismia varmistaakseen, että populaation parhaat yksilöt selviytyvät muuttumattomina seuraavalle sukupolvelle. [1.]

2.3 Algoritmin toimintavaiheet

Geneettisen algoritmin populaatio alkaa useimmiten satunnaisesti generoituna. Tämän jälkeen populaatiota muokataan toistettavilla operaatioilla [1].

Ensimmäiseksi yksilöiden soveltuvuus mitataan hyödyntäen soveltuvuusfunktiota (engl. *fitness function*). Soveltuvuusfunktio ottaa parametrina sisään yksilön ja palauttaa soveltuvuusarvon, joka ilmaisee yksilön suorituskyvyn tai soveltuvuuden kyseessä olevan ongelman ratkaisemiseen. [3.]

Kun jokaisella yksilöllä on oma soveltuvuusarvonsa, voimme suorittaa suosittavien yksilöiden valinnan seuraavan sukupolven vanhemmiksi. Valintatapoja on monenlaisia, mutta yhteinen tekijä niiden välillä on usein elitismi, eli parhaiden yksilöiden suosiminen. Osa valintamenetelmistä voi myös valita huonompia yksilöitä satunnaisesti ylläpitääkseen populaation monimuotoisuutta. Valittuja yksilöitä käytetään vanhempina risteytysoperaatioita tehdessä. Risteytyksen tarkoituksena on tuottaa uusia yksilöitä yhdistämällä vanhempien geenejä tekijäinvaihdunnan avulla ja korvata populaation huonot yksilöt. [3.] Kuvassa 1 on havainnollistettu tekijäinvaihdunta käyttäen kahta pistettä.



Kuva 1. Tekijäinvaihdunta käyttäen kahta pistettä.

Uusien yksilöiden geenejä myös mutatoidaan eli muutetaan satunnaisesti, jotta algoritmi voisi etsiä mahdollisia ratkaisuja laajemmalla alueella ja ylläpitää populaation monimuotoisuutta. [1.]

Näitä operaatioita toistetaan, kunnes algoritmi tai sen tuottamat yksilöt täyttävät jonkin lopetusehdon tai kunnes algoritmi pysäytetään manuaalisesti. Lopetusehtona voi toimia esimerkiksi sukupolvien maksimimäärä, tietyn soveltuvuusarvon rajan ylittäminen tai yksilöiden suorituksen manuaalinen tarkistus. [2.] Kuvassa 2 on esitelty tavallisen geneettisen algoritmin toimintavaiheet pseudokoodina.

Generoi populaatio satunnaisesti;

Kun lopetusehto ei ole täytetty:

Mittaa yksilöiden soveltuvuus;

Valitse parhaat yksilöt;

Risteytä hyvät yksilöt ja korvaa huonot yksilöt uusilla;

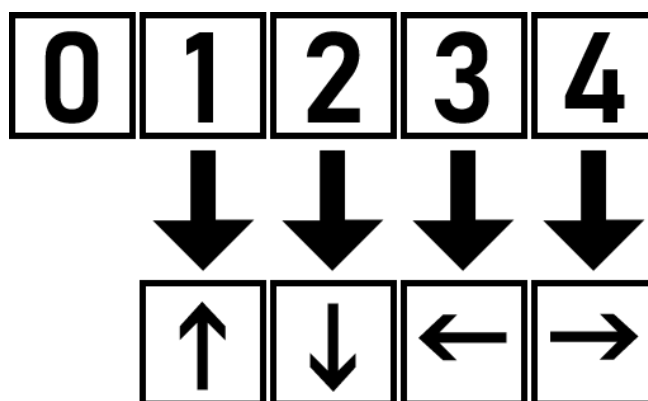
Mutatoitaisia yksilöitä;

Kuva 2. Geneettisen algoritmin yleiset toimintavaiheet.

3 Soveltaminen peleissä liikkumiseen

3.1 Ohjauskomennot

Pelihahmon ohjaus geneettisen algoritmin avulla käytännössä tapahtuu tuottamalla ohjauskomentoja peliin. Eräs yksinkertainen tapa tuottaa ohjauskomentoja on luoda geneettinen algoritmi, joka tuottaa sarjan numeroita tietyltä väliltä. Esimerkiksi jos haluaisimme algoritmin pystyvän liikkumaan neljään eri suuntaan tai olemaan liikkumatta, voisimme toteuttaa tämän tuottamalla sarjan numeroita väliltä 0–4. Tämän jälkeen numerot kartoitetaan ohjauskomennoiksi seuraamalla tiettyä kaavaa. Esimerkiksi numero 0 voisi olla tyhjä komento, josta ei seuraa mitään toimenpidettä pelissä. Numerot 1–4 voisivat vastata liikkumista eteen, taakse, vasemmalle ja oikealle. Numeroiden yhteys komentoihin on havainnollistettu kuvassa 3.



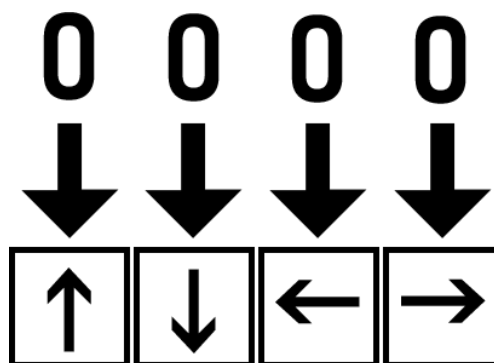
Kuva 3. Numeroiden kartoitus liikkumissuuntien ohjauskomentoihin.

Sarja ohjauskomentoja suoritetaan pelissä yksi kerrallaan aikajärjestyksessä. Komentojen välinen aika riippuu kyseisen pelin liikkumiskomentojen päivitystaajuudesta. Tietyt pelit ottavat vastaan liikkumiskomentoja jokaisella kuvanpäivityksellä, kun taas toiset pelit voivat ottaa komentoja vastaan muuttumattomalla taajuudella, esimerkiksi 60 kertaa sekunnissa, riippumatta siitä, kuinka usein kuva päivitetään pelaajalle.

Geneettisen algoritmin tuottamien geenien ja ohjauskomentojen yksi yhteen kartoitus kuitenkin tarkoittaa sitä, että algoritmi ei voi suorittaa useamman komennon yhdistelmää samanaikaisesti. Useassa pelissä pelaajat voivat hyödyntää useamman ohjauskomennon yhdistelmää muun mu-

assa liikkuaakseen vinoittain. Yhdistämällä ohjauskomennon eteenpäin liikkumiselle joko ohjauskomennon vasemmalle tai oikealle liikkumisen kanssa pelaajahahmo voi liikkua vinoittain 45 asteen kulmassa.

Yksi mahdollinen ratkaisu useamman samanaikaisen ohjauskomennon toteuttamiseen on korvata geenin sisältämä yksittäinen numero bittipeitteellä. Neljän liikkumiskomennon yhdistelmän kuvaamiseen voimme käyttää neljän bitin peitettä, jossa jokaisen bitin arvo vastaa yhden ohjauskomennon arvoa. Esimerkiksi peite 0000 tarkoittaa, että mitään ohjauskomentoa ei suoriteta, kun taas peite 1111 tarkoittaa, että kaikki neljä ohjauskomentoa suoritetaan samanaikaisesti. Toisin sanoen algoritmi ei tuota yksittäisten ohjauskomentojen sarjaa, vaan sarjan useamman ohjauskomennon yhdistelmiä. Bittipeitteen yksittäisten bittien yhteys komentoihin on havainnollistettu kuvassa 4.



Kuva 4. Bittipeite liikkumissuuntien ohjauskomennoille.

3.2 Mutaatio

Bittipeitteen hyödyntäminen tulee ottaa huomioon mutaatioita tehdessä. Geenien ollessa yksittäisiä binääriarvoja mutaatio yksinkertaisesti kääntää binääriarvon nolasta ykköseksi tai toisin päin. Koko arvon kääntäminen toimii yksittäisen binääriluvun kanssa hyvin, koska mahdollisia arvoja on ainoastaan kaksi ja mutaatiolla on mahdollisuus johtaa kumpaan tahansa arvoon.

Myös nelibittisen peitteen tapauksessa koko peitteen kääntäminen vaihtaa arvoa ainoastaan kahden arvon välillä. Esimerkiksi peitteen 1010 käänteinen peite on 0101 ja peitteen kääntäminen uudestaan johtaa takaisin alkuperäiseen arvoon. Neljän bitin pituisella peitteellä on 2^4 , eli 16 mahdollista arvoa. Ihanteellisesti algoritmi pystyisi mutatoimaan peitettä vapaasti siten, että jokaisen 16 arvon lopputulos olisi mahdollinen riippumatta peitteen alkuperäisestä arvosta.

Tämän takia jokaisella geenillä ei tulisi olla ainoastaan yhtä satunnaista mahdollisuutta mutaatioitua, vaan bittipeitteen jokaisella bitillä on oma mahdollisuutensa mutaatioitua. Käytännössä mutaatiot toteutetaan bitittäisiä operaatioita hyödyntäen. Kuva 5 demonstroi yksittäisen liikkumiskomennon kääntämistä bittipeitteessä hyödyntäen C++-kielen bitittäisiä operaattoreita.

```

1. // Liikkumiskomennot
2. #define FORWARD 0x1000
3. #define BACK    0x0100
4. #define LEFT    0x0010
5. #define RIGHT   0x0001
6.
7. // Esimerkki bittipeite
8. int mask = 0x1111;
9.
10. // FORWARD-liikkumiskomennon arvon kääntäminen
11. if ((mask & FORWARD) == FORWARD)
12. {
13.     mask &= ~FORWARD;
14. }
15. else
16. {
17.     mask |= FORWARD;
18. }

```

Kuva 5. Yksittäisen liikkumiskomennon kääntäminen bittipeitteessä bitittäisillä operaatioilla.

3.3 Soveltuvuuden mittaaminen

Yksilöiden soveltuvuuden mittaaminen voi yksinkertaisimmillaan olla esimerkiksi etäisyys suoritettavan tason alku- tai loppupisteestä liikkumiskomentosarjan suorituksen jälkeen. Jos tasossa on seiniä tai muita esteitä, yksinkertainen etäisyyden mittaaminen voi johtaa ongelmiin soveltuvuusarvon paikallisten maksimiarvojen kanssa, jolloin yksilöt jäävät jumiin estettä vasten sen ympäri kiertämisen sijaan.

Soveltuvuusfunktion huolimaton määrittäminen voi johtaa odottamattomaan tai ei haluttuun käytökseen. Esimerkiksi jos algoritmin on mahdollista hävitä peli ja yksilöt saavat lisää soveltuvuus pisteitä käytetyn ajan perusteella, on mahdollista, että algoritmin on kannattavampaa hävitä peli mahdollisimman nopeasti kentän läpäisyn sijaan minimoidakseen käytetty aika ja maksimoidakseen soveltuvuusarvo.

3.4 Determinismi

Determinismi, eli idea ennalta määrätystä tapahtumien kulusta, on käsite, jota käytetään kuvaamaan uudelleen tuotettavuutta ja tapahtumien johdonmukaisuutta [4]. Esimerkiksi pelien fysiikkamoottoreita kutsutaan deterministisiksi, jos fysiikkaobjektien sama alkutilanne johtaa aina samaan objektien loppuasetelmaan.

Determinismi on erityisen tärkeää peleissä, joissa taitavat pelaajat voivat ennustaa tapahtumien kulkua alkutilanteen mukaan, kuten esimerkiksi biljardissa. Taitava pelaaja osaa ennustaa, mihin biljardipallot tulevat päätymään riippuen siitä, missä kulmassa ja millä voimakkuudella niitä lyödään. Tämän takia hyvässä biljardipelissä pallojen tulisi aina käyttäytyä samalla tavalla, jotta pelaaja ei tunne oloaan huijatuksi. Saman pallojen alkuasetelman ja saman lyönnin toteutus useamman kerran pitäisi aina tuottaa sama lopputulos.

Liikkumismekaniikoiden tapauksessa saman liikkumiskomentosarjan suorittaminen useamman kerran tulisi tuottaa sama loppusijainti pelaajahahmolle. Käytännössä tämä tarkoittaa, että liikkumisessa ei tulisi käyttää satunnaisesti tuotettuja lukuja. Myös mahdolliset virheet pelin päivitystasaajuuden vaihtelusta, verkkoviiveen muutoksesta, verkkopakettien pudotuksesta tai liukulukujen laskutoimituksista tulisi minimoida.

Uudelleen tuotettavuus on välttämätöntä geneettisen algoritmin toiminnalle. Jos liikekomentosarjan tuottama lopputulos on satunnainen sijainti, myös kyseisen yksilön soveltuvuusarvo tulee olemaan satunnainen. Yksilöiden soveltuvuuden tietäminen on pakollista elitismien harjoittamiselle ja siten geneettisen algoritmin toiminnalle yleensä. Koska liikekomennot suoritetaan ajan myötä sarjassa yksi toisensa jälkeen, jopa hyvin pienet satunnaisuudet liikkumismekaniikoissa voivat johtaa suuriin satunnaisiin muutoksiin lopputuloksessa niin sanotun perhosvaikutuksen myötä.

3.5 Hyvät ja huonot puolet

Geneettinen algoritmi ei varsinaisesti opi pelaamaan peliä samalla tavalla kuin esimerkiksi neuroverkot. Geneettisen algoritmin tuottamat liikkumiskomentosarjat toimivat ratkaisuna ainoastaan kyseiseen pelitasoon tai ongelmaan, jota varten ne tuotettiin. Koska jokaiselle tasolle täytyy etsiä oma ratkaisu ja ratkaisun löytäminen vaatii paljon laskenta-aikaa, geneettinen algoritmi ei sovi reaaliaikaiseen navigointiin.

Perinteisen tekoälyn kannalta polun löytäminen peleissä on suhteellisen helposti ratkaistavissa oleva ongelma hyödyntäen olemassa olevia polunetsintäalgoritmeja, kuten A*-algoritmia tai Dijkstran algoritmia. Tekoälyagentti, joka kykenee navigoimaan peliympäristössä käyttäen samoja liikkumismekaniikoita kuin ihmispelaajat voi kuitenkin olla erittäin vaikea toteuttaa tietyissä peleissä. Geneettinen algoritmi kiertää tekoälyohjelmoinnin haasteet laskenta-ajan kustannuksella.

4 Käytännön sovellus Source-pelimoottorin Surf-pelimuodossa

4.1 Source

Source on Valve Corporationin kehittämä 3D-pelimoottori, joka luotiin korvaamaan heidän aikaisempi GoldSource-pelimoottorinsa. Ensimmäinen Source-pelimoottorilla julkaistu peli oli Counter-Strike: Source, joka julkaistiin 1.11.2004. [5, 6.] Tämän jälkeen Source-pelimoottorilla julkaistiin muun muassa pelit Half-Life 2 (kuva 6), Team Fortress 2 ja Counter Strike: Global Offensive [7, 8, 9].

Source-pelimoottorin julkaisun jälkeen monet peliyhtiöt ja pelaajayhteisöt ovat käyttäneet moottoria uusien pelien ja modien luontiin [5].



Kuva 6. Kuvakaappaus Valve Corporationin pelistä Half-Life 2, joka on yksi suosituimmista Source-pelimoottorilla luoduista peleistä.

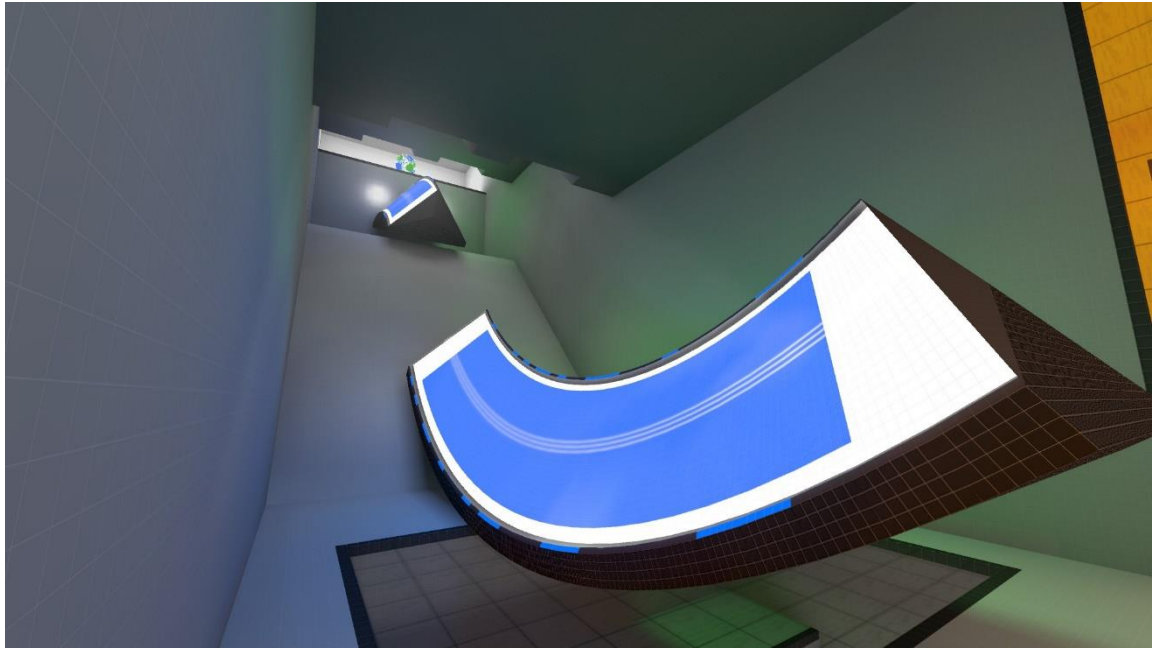
4.2 Surf

Surf on Counter-Strike-peliyhteisön luoma pelimuoto, joka sai alkunsa vuonna 2004 kun Charlie Joyce loi ensimmäiset pelikentät, jotka olivat omistettuja Surf-pelimekaniikoita varten. Surf-pelimuodon keskeinen mekaniikka on *“surffaus”* (engl. *surfing*) eli kallistettuja pintoja pitkin liukuminen. Surffaus hyödyntää GoldSource- ja Source-pelimoottorien liikkumismekaniikoita, erityisesti sitä, kuinka moottorit hoitavat pelaajien liikkumisen kallistetulla lattialla. Normaalisti pelaajahahmot liukuvat alas jyrkästi kallistetulla pinnalla, mutta pitämällä pohjassa liikkumisnappia kallistettua pintaa kohti pelaajahahmo voi sen sijaan liukua eteenpäin pintaa pitkin. [10.]

Toinen tärkeä pelimekaniikka Surf-pelimuodossa on *“airstrafe”*-mekaniikka eli pelaajan vapaus kääntyä ilmassa ja samalla muuttaa nopeutensa horisontaalisen komponentin suuntaa. Pelaajan kyky muuttaa nopeuttaan ilmassa on jäännös siitä, kuinka liikkumisen kiihtyvyys toteutettiin Quake-pelimoottorissa, johon Valven GoldSource-pelimoottori pohjautuu. [11, 12.]

Monimutkaisien liikkumismekaniikoidensa takia Surf-pelimuoto vaatii sorminäppäryyttä tiettyjen näppäimistönäppäimien ja hiiren liikkeen yhdistelmien toteuttamiseen. Pelimuodon vaativuuden lisäksi pelaajat usein kilpailevat toisiaan vastaan ja yrittävät läpäistä kenttiä mahdollisimman lyhyessä ajassa.

Kuvassa 7 on kuvakaappaus Surf-pelikentästä, jossa on näkyvissä sinisiä kolmionmuotoisia ramppeja, joita pitkin pelaajien on tarkoitus liukua. Pelitason aloitussijainti on kuvassa ylävasemmalla. Ramppien välillä on rako, jonka yli pelaajien täytyy päästä hyödyntämällä pelimekaniikoita ilmassa liikkumiseen.



Kuva 7. Kuvakaappaus Surf-pelikentästä *surf_It_omnific*.

4.3 Sovellettavuus

Kaikista Source-pelimoottorilla luoduista peleistä minulla on, sekä pelaajana että ohjelmoijana, eniten kokemusta Team Fortress 2 -pelistä (TF2). Tämän takia päätin käyttää TF2-peliä käytännön sovelluksen alustana.

Pelaajien liikkumismekaniikat ovat deterministisiä TF2-pelin Surf-pelimuodossa, eikä niissä ole huomattavaa satunnaisuutta. Ohjelmointia varten hyödynnettiin Source-pelimoottorille luotua SourceMod-rajapintaa. SourceMod-projekti sisältää oman SourcePawn-ohjelmointikielen, kääntäjän ja ohjelmointirajapinnan, joka sallii Source-pelipalvelimen toiminnallisuuden muokkaamisen. Pelaajan liikekomentojen sekä muiden tarvittavien operaatioiden suorittaminen on mahdollista tämän rajapinnan kautta. [13.]

4.4 Ohjauskomennot

Ohjauskomentoina toimii viisi binääriarvoa sekä kaksi liukulukuarvoa.

Binääriarvot vastaavat liikkumiskomentoja eteenpäin, vasemmalle ja oikealle liikkumiseen sekä komentoja ylöspäin hyppäämiseen ja kyykkyyntymiseen. TF2-pelissä on useampia muita komentoja, joita pelaajat voivat käyttää, kuten taaksepäin käveleminen, mutta nämä komennot eivät ole välttämättömiä Surf-pelimuodon pelaamiseen ja tulisivat lisäämään geneettisen algoritmin hakualuetta.

Ensimmäinen liukuluvuista vastaa pelaajan pyörimistä pysty akselin ympäri vasta- tai myötäpäivään. Source-pelimoottorissa kiertymiskulman arvo pysty akselin ympäri on rajoitettu välille ± 180 astetta ja arvo kiertyy ympäri toiseen ääripäähän mennessään tämän arvon ulkopuolelle.

Toinen liukuluku puolestaan ohjaa pelaajan kameran kallistamista ylös tai alas. Tämä arvo on rajoitettu Source-pelimoottorissa välille ± 89 , eikä sen arvo kierry ympäri vaan pysähtyy raja-arvon kohdalle. Pelikameran kallistuskulma ei tässä tapauksessa vaikuta pelaajahahmon liikkumiseen, joten kallistuksen arvo on rajoitettu sovelluksessa välille ± 30 astetta, jotta algoritmin pelisuoritusta olisi mukavampi katsoa.

Liikkumiskomentojen binääriarvot ilmaisevat kyseisen liikkumiskomennon olevan joko päällä tai pois päältä. Liikkumiskomentojen tallennuksessa ja muokkauksessa hyödynnetään bittipeitettä. Liukuluvut puolestaan ovat delta-arvoja, ja ne ilmaisevat kulman suuruuden muutosta yhden päivituksen aikana. Molemmat liukuluvut ovat rajoitettuja välille ± 2.5 astetta per päivitys.

4.5 Hakualue

TF2-pelissä palvelimen ja samalla pelaajan liikkumiskomentojen päivitys tapahtuu 66.66... kertaa sekunnissa eli 15 millisekunnin välein. Tämän seurauksena geneettisen algoritmin hakualue kasvaa erittäin nopeasti.

Liukulukuarvojen hakualuetta on hankala kartoittaa arvojen ollessa jatkuvia ja delta-arvojen ollessa satunnaisesti arvottuja raja-arvojen väliltä. Source-pelimoottori käyttää kulmien liukuluvuissa 16 desimaalipaikan tarkkuutta, joten jopa arvojen -2.5 ja $+2.5$ välillä mahdollisia lukuja on arviolta $5 \cdot 10^{16}$.

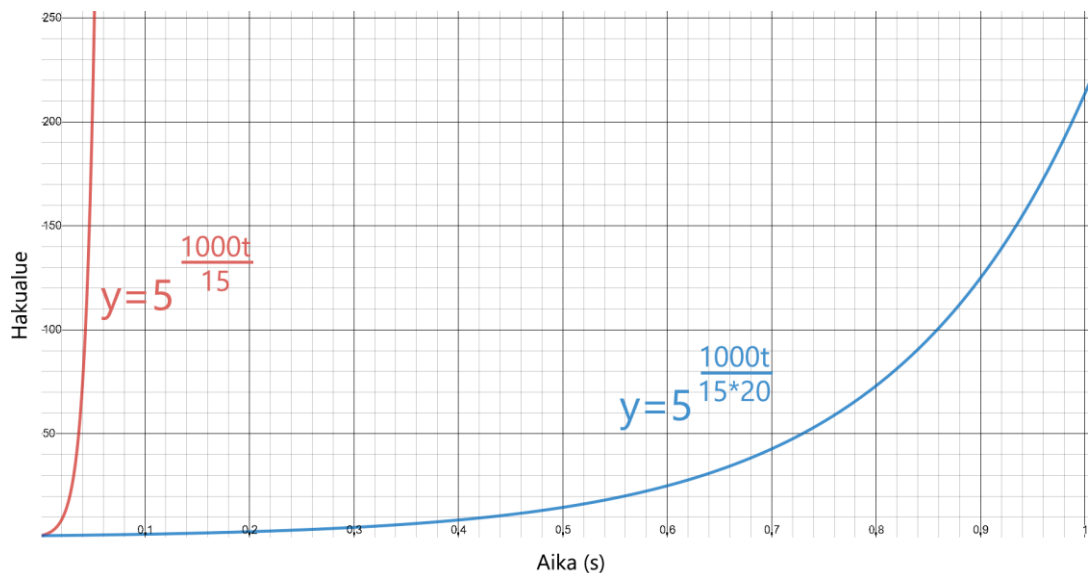
Binääriarvojen hakualue on helpommin hahmoteltavissa. Käytössä on viisi komentoa, joista jokainen voi olla joko päällä tai pois päältä. Jos haluaisimme tuottaa sarjan komentoja tietylle aikajaksonalle, hakualue olisi tällöin $5^{\frac{1000t}{15}}$, jossa t on aikajakson pituus sekunneissa. Hakualueen arviosta

näkee heti, että alue kasvaa eksponentiaalisesti verrattuna haetun ratkaisun aikajakson pituuteen. Tämän takia lyhyet tasot ovat huomattavasti helpompia ratkaista kuin pidemmät.

Hakualueen optimointiin voidaan hyödyntää komentojen toistamista useamman päivityksen ajan, sen sijaan, että jokaiselle päivitykselle tuotettaisiin uusi yhdistelmä komentoja. 20 päivitystä eli 300 millisekuntia tuntui sopivan tarkalta aikaväliltä ohjauskomentojen muutostaajuudelle

Surf-pelimuodossa. Optimoinnin jälkeen ohjauskomentojen binääriarvojen hakualue on $5^{\frac{1000t}{15 \cdot 20}}$.

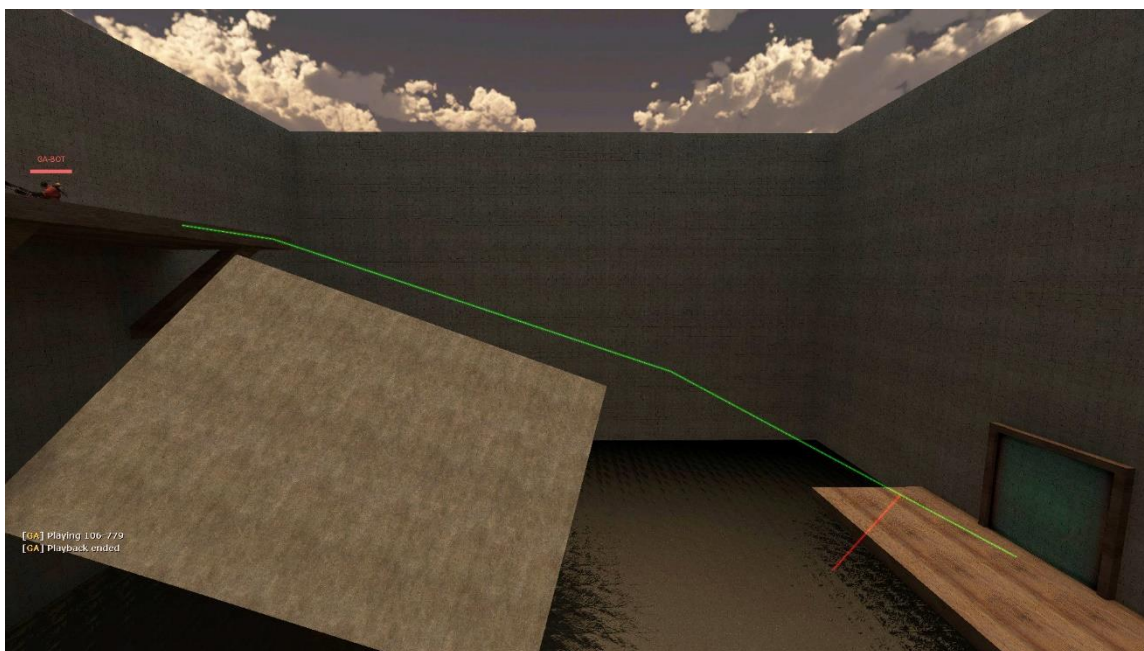
Näiden hakualueiden kasvunopeuden ero on nähtävissä kuvassa 8. Liikkumiskomentojen toistaminen myös tekee algoritmin tuottamasta liikkeestä huomattavasti sulavampaa, koska pelikameran ja pelaajahahmon liikkumissuunnat vaihtuvat harvemmin.



Kuva 8. Ei-optimoidun ja optimoidun binääriarvojen hakualueen eksponentiaalinen kasvu.

4.6 Soveltuvuuden mittaaminen

Soveltuvuuden mittaaminen Surf-pelimuodossa perustuu liikekomentosarjan seurauksena matkustetun etäisyyden mittaamiseen. Matkustetun etäisyyden mittaamista varten hyödynnetään useasta pisteestä muodostettua viivaa, joka alkaa tason alusta ja kulkee tason loppuun asti. Kuvassa 9 on havaittavissa usean pisteen kautta kulkeva vihreä viiva, joka kulkee tason alkupisteestä, kuvassa ylävasemmalla, tason loppupisteeseen, kuvassa alhaalla oikealla.



Kuva 9. Pelitasossa soveltuvuuden mittaamiseen käytetty viiva (vihreä).

Kuvan oikeassa alakulmassa on myös havaittavissa lyhyempi punainen viiva, joka lähtee vihreästä viivasta suorakulmassa ja päättyy lattialla sijaitsevan veden pintaan. Punaisen viivan alimmainen kohta kuvastaa pistettä avaruudessa, jossa algoritmin tuottaman komentosarjan suoritus päättyi. Punaisen ja vihreän viivan kohtisuora yhteyspiste kuvastaa vihreän viivan lähintä pistettä pelaajahahmoon algoritmin suorituksen päättyessä. Tällöin punaisen viivan pituus on pelaajahahmon etäisyys vihreästä viivasta suorituksen lopussa.

Algoritmin suoritus voi päättyä kolmesta eri syystä. Tässä tapauksessa pelaajahahmo osui lattialla olevaan veteen, jonka seurauksena pelaaja häviää tason ja pelaajahahmo siirretään takaisin tason alkupisteeseen. Kaksi muuta mahdollista syytä suorituksen päätökseen ovat tason loppupisteen välittömään läheisyyteen pääsy tai suoritukselle varatun aikarajan ylittyminen.

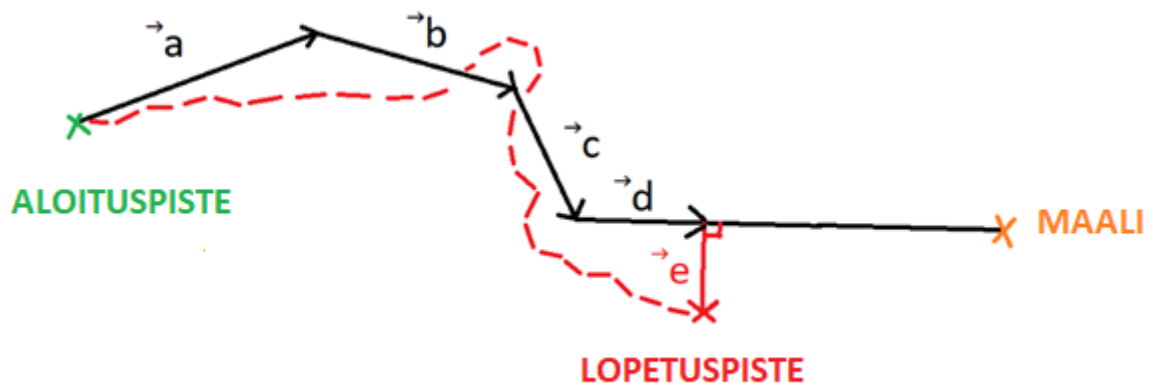
Yksilön soveltuvuusarvo määritetään mittaamalla etäisyys vihreää viivaa pitkin siihen pisteeseen asti, joka on lähimpänä suorituksen päätöspistettä. Tästä arvosta vielä vähennetään etäisyys pois päin vihreästä viivasta eli punaisen viivan pituus. Soveltuvuusfunktion tarkoituksena on toisin sanoen optimoida, kuinka pitkälle yksilöt pääsevät vihreän viivan suuntaan ja minimoida suorituksen lopussa mahdollinen etäisyys vihreästä viivasta pois päin.

Yksilöt saavat myös lisää soveltuvuus pisteitä säästetystä ajasta. Jokaiselle tasolle on manuaalisesti määritelty aikaraja, jonka jälkeen algoritmin suoritus päättyy, jos se ei ole jo päättynyt

muusta syystä. Jos yksilö pääsee tason loppuun asti, se saa pisteitä aikarajan ja suorituksen todellisen keston välisestä erotuksesta.

Jos yksilö ei pääse loppuun asti, se saa silti pisteitä aikarajan alituksesta, mutta pistemäärä kerrotaan pienellä kertoimella. Lisäpisteiden antaminen ajasta, vaikka yksilö ei läpäise tasoa, kannustaa algoritmia läpäisemään tasoa nopeammin. Jos esimerkiksi kaksi algoritmin tuottamaa yksilöä häviää tason samassa kohdassa avaruutta, se, joka pääsi siihen pisteeseen nopeammin, on kuitenkin soveltuvuuden kannalta parempi yksilö. Häviämisen tapauksessa on kuitenkin tärkeää pitää mielessä, että yksilöt eivät saa liikaa pisteitä säästetystä ajasta verrattuna matkustettuun etäisyyteen. Jos nopeasti häviäminen johtaisi suurempaan soveltuvuusarvoon kuin tietyn etäisyyden matkustaminen, on mahdollista, että algoritmi jumittuu paikalliseen soveltuvuuden maksimiin ja yrittää aina hävitä mahdollisimman nopeasti tason läpäisemisen sijaan.

Kuvassa 10 on havainnollistettu yksinkertaistettu kaksiulotteinen esimerkkisuoritus pelitasossa. Pelaajahahmon liikerata on merkitty punaisella katkoviivalla. Soveltuvuuden mittaamiseen käytettävä viiva on merkitty mustalla ja se koostuu useammasta vektorista. Etäisyys viivasta suorituksen lopetuspisteeseen on havainnoitu punaisella vektorilla.



Kuva 10. Kaksiulotteinen esimerkki yksilön liikkeestä ja soveltuvuuden mittaamisesta.

Kyseisen suorituksen tapauksessa soveltuvuus voitaisiin laskea seuraavalla kaavalla:

$$\text{soveltuvuus} = |\vec{a}| + |\vec{b}| + |\vec{c}| + |\vec{d}| - |\vec{e}| \quad (1)$$

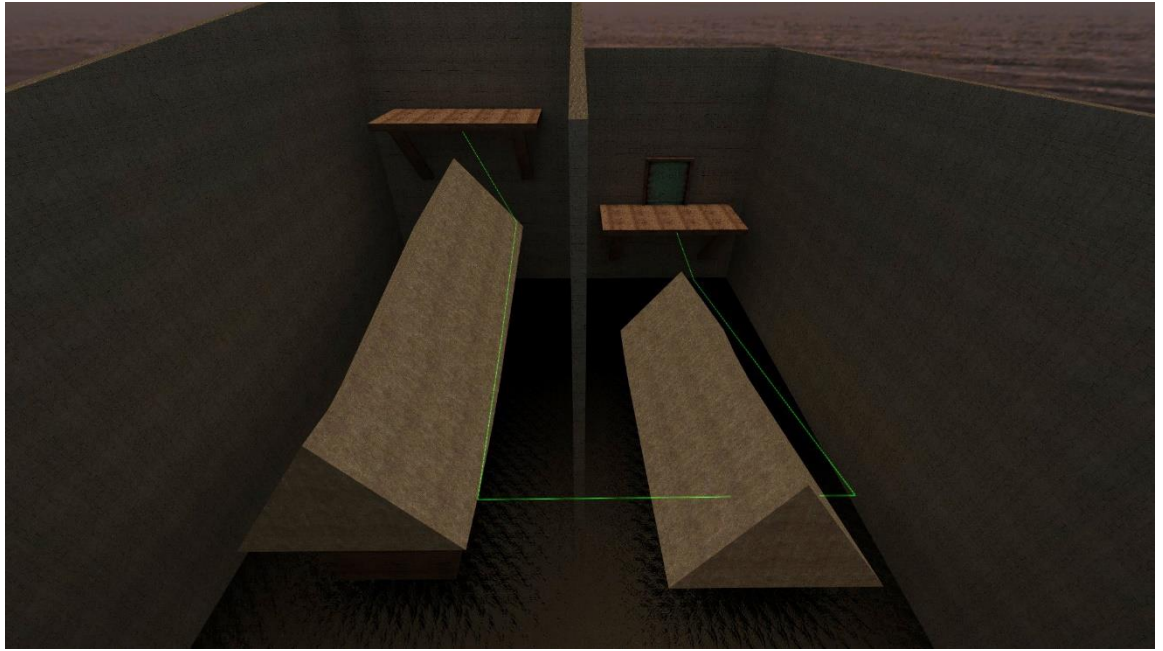
Tämän lisäksi soveltuvuuteen voidaan lisätä pisteitä säästetystä ajasta seuraavalla kaavalla:

$$\text{soveltuvuus} += (\text{aikaraja} - \text{käytetty aika}) * \text{skaala} \quad (2)$$

Ajasta annettavien pisteiden skaalan tarkoituksena on välttää aiemmin mainittu mahdollinen ongelma, joka voisi kannustaa algoritmia häviämään mahdollisimman pian. Myös alkuperäisen soveltuvuusarvon kaavan tapauksessa voisimme käyttää skaalaa tai eri laskentatapoja vektorin \vec{e} muokkaamiseen. Esimerkiksi vektoria \vec{e} voisi skaalata joko pienemmällä tai suuremmalla kertoimella, jotta algoritmi priorisoi etäisyyttä viivasta enemmän tai vähemmän. Voisimme myös skaalata vektorin komponentteja erikseen, painostaakseen esimerkiksi etäisyyttä pystysuunnassa enemmän kuin vaakasuunnassa tai antaakseen lisäpisteitä, jos yksilö on viivan yläpuolella, koska korkeus on usein haluttu ominaisuus tason suorituksen kannalta.

Tässä käytännön sovelluksessa soveltuvuusfunktio ei ota huomioon seiniä tai viivan näkyvyyttä haettaessa lähintä pistettä soveltuvuusviivalta. Muissa peleissä tai pelimuodoissa on mahdollista hyödyntää ehtoa, jonka takia yksilö ei saa soveltuvuuspisteitä mahdollisten ongelmakohtien välttämiseksi, jos näköyhteyttä viivalle ei ole. Tässä tapauksessa näkyvyyden vaatiminen soveltuvuuden mittaamiseksi on ongelmallista Surf-tasojen kenttägeometrian luonteen takia. Suurin ongelma muodostuu kentissä olevista rampeista, joiden alapuolelle pelaajahahmo voi pudota. Tällaisessa tapauksessa objektiivisesti parempi yksilö, joka pääsee kentässä pidemmälle, mutta sattuu häviämään tason rampin alapuolella kohdassa, josta ei ole näköyhteyttä viivalle, saisi vähemmän pisteitä kuin lyhemmälle päässyt yksilö, joka ei sattunut häviämään kohdassa, josta viivaa ei näe.

Koska soveltuvuusfunktio ei havaitse seiniä tai muita esteitä, on tärkeää pitää mielessä tasojen mahdolliset ongelmakohdat suunniteltaessa soveltuvuusviivaa. Esimerkiksi kuvassa 11 mahdollinen ongelmakohta on tason keskellä oleva seinä. Jos yksilö häviää tason seinän läheisyydessä, on mahdollista, että lähin piste soveltuvuusviivalta haetaan seinän toiselta puolelta.

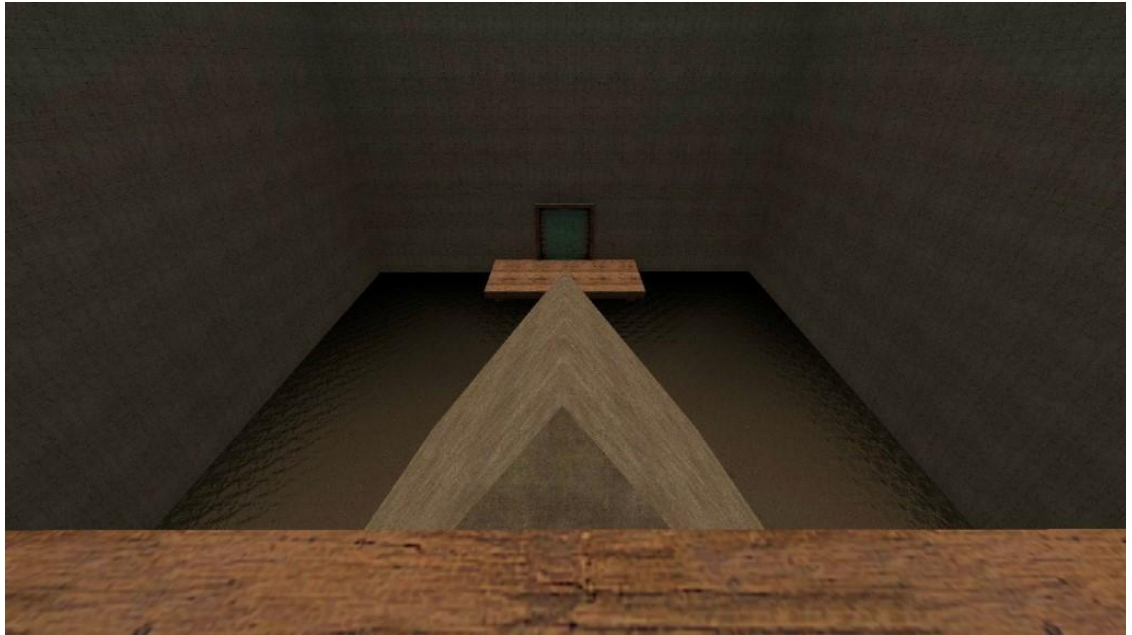


Kuva 11. *surf_beginner*-pelikentän kolmas taso ja soveltuvuusviiva.

4.7 Tulokset

Käytännön sovelluksen toiminnallisuuden ohjelmointi onnistui SourceMod-rajapintaa hyödyntäen ilman suurempia ongelmia. Sovelluksen lähdekoodi löytyy liitteestä 1.

Sovelluksen testauksen kohteena käytettiin pelikenttää nimeltä "*surf_beginner*". Kuten kentän nimestä voi päätellä, kyseinen kenttä on suunniteltu kokemattomille pelaajille, jotka ovat vasta aloittaneet pelimuodon pelaamisen. Kenttä sisältää useita helposti läpäistävissä olevia tasoja, ja sen katsottiin olevan hyvä todiste sovelluksen toimivuudesta. Kuvassa 12 on nähtävissä pelikentän ensimmäinen taso.

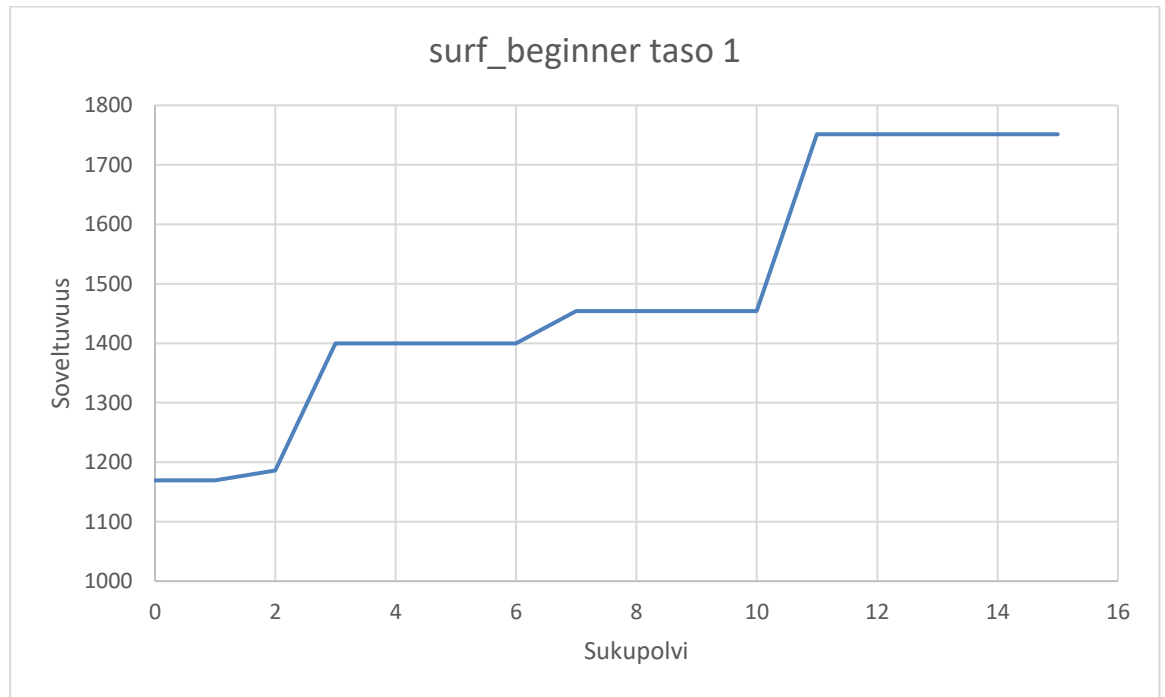


Kuva 12. *surf_beginner*-pelikentän ensimmäinen taso kuvattuna tason aloitustasanteelta.

Sovellus on tähän mennessä onnistunut läpäisemään pelikentän neljä ensimmäistä tasoa. Video-kuva algoritmin suorituksesta tasoissa löytyy liitteestä 2. Algoritmi löysi kuudessa minuutissa ensimmäisen ratkaisun, joka pääsi ensimmäisen tason loppuun asti. Toisen tason ensimmäinen ratkaisu löytyi noin kahdeksassa tunnissa, kolmannen tason noin kymmenessä tunnissa ja neljännen tason noin 17 tunnissa. Ensimmäiseen onnistuneeseen tason läpäisyyn vaadittu laskenta-aika kasvaa eksponentiaalisesti, kun tason pituus tai monimutkaisuus kasvaa.

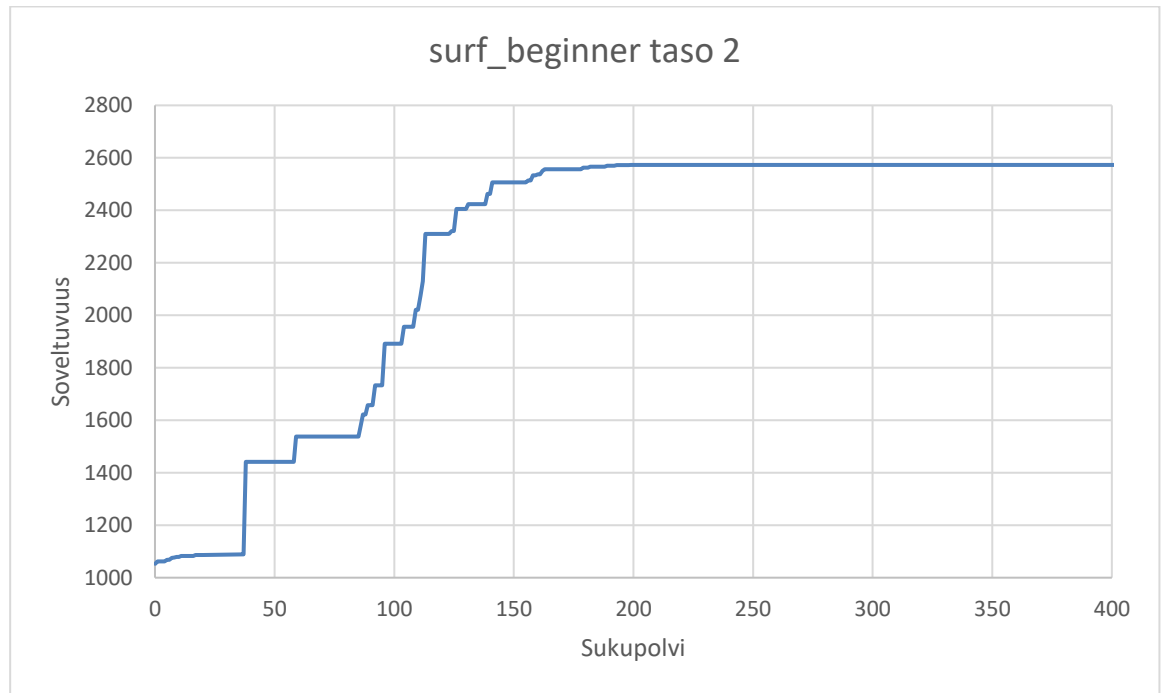
Kahden ensimmäisen tason ajalta kerättiin jokaisen sukupolven parhaan yksilön soveltuvuusarvo, jotta soveltuvuusarvon kasvua voitaisiin havainnoida kaavamaisesti. Algoritmin suoritusta on myös hyvä vertailla käyttäen kahta ensimmäistä tasoa näiden eroavien laskenta-aikavaatimuksien takia.

Kuvassa 13 näkyy vaaka-akselilla ensimmäisen tason aikana tuotetun populaation sukupolvet ja pystyakselilla jokaisen sukupolven parhaan yksilön soveltuvuusarvo. Sukupolvi 0 on satunnaisesti tuotettu, ja sukupolven 11 paras yksilö oli ensimmäinen, joka pääsi tason loppuun saakka. Sukupolvien 0 ja 10 välillä soveltuvuusarvo kasvaa saamaisesti, ja sukupolven 11 kohdalla arvo nousee jyrkästi, koska algoritmi saa lisää pisteitä tason loppuun asti pääsemisestä. Geneettisen algoritmin ratkaisujen odotetaan paranevan logaritmisesti, mutta algoritmi läpäisee ensimmäisen tason erittäin nopeasti, ja datapisteiden vähäisen määrän takia soveltuvuuden paranemisnopeutta on vaikea arvioida.



Kuva 13. *surf_beginner*-pelikentän ensimmäisen tason aikana generoitujen sukupolvien parhaiden yksilöiden soveltuvuusarvot.

Kuvassa 14 puolestaan näkyy toisen tason aikana tuotetut sukupolvet ja niiden parhaiden yksilöiden soveltuvuusarvot. Soveltuvuusarvon logaritminen kasvu on helpommin havaittavissa toisessa tasossa, koska sen läpäisemiseen vaadittu aika ja sukupolvien määrä ovat huomattavasti suurempia kuin ensimmäisessä tasossa ja tätä myötä myös datapisteitä on enemmän. Soveltuvuusarvo kasvaa logaritmisesti sukupolvien 0 ja 400 välillä, ja ensimmäinen yksilö, joka pääsee tason loppuun asti, löytyi sukupolvesta 713 (ei kuvattu), noin kahdeksan tunnin laskenta-ajan jälkeen. Soveltuvuusarvon kasvu lähes pysähtyy sukupolven 200 jälkeen, kunnes sukupolven 713 paras yksilö onnistui laskeutumaan tason lopputasanteelle.



Kuva 14. *surf_beginner*-pelikentän toisen tason aikana generoitujen sukupolvien parhaiden yksilöiden sovellettuusarvot.

5 Jatkokehitys

5.1 Vaihtuva liikekomentojen toistamistaajuus

Liikekomentojen toistaminen useamman päivityksen ajan on erittäin tehokas menetelmä hakualueen optimointia varten, mutta se voi mahdollisesti estää algoritmia suorittamasta tarkkoja liikkeitä hankalampien tasojen läpäisyä varten. Tämän takia liikekomentojen toistamistaajuutta olisi hyvä voida lyhentää, kun algoritmi pääsee vaiheeseen, jossa se ei enää paranna suoritustaan huomattavasti nykyisellä toistamistaajuudella.

Kim ja de Weck demonstroivat topologiseen optimointiin liittyvässä julkaisussaan, kuinka ajan myötä ratkaisun resoluutiota kasvattava adaptiivinen geneettinen algoritmi voi päihittää sekä laskenta-ajan että ratkaisun laadun suhteen tavallisen geneettisen algoritmin, joka aloittaa suorituksen hakemisen suurimmalla resoluutiolla heti alusta lähtien [14]. Ratkaisun resoluution kasvattaminen topologiaongelmissa on verrattavissa Surf-pelimuodossa liikekomentojen toistamisajan lyhentämiseen, jolloin algoritmin suoritus koostuu useammista lyhyemmistä liikekomentojen päätöksistä.

Toiminnallisuuden lisäys sovellukseen itsessään liikekomentojen toistamistaajuuden muutokselle on helposti toteutettavissa. Suurempi ratkaistava ongelma on heuristiikka, jonka perusteella havaitaan, milloin algoritmin suoritus ei enää parane huomattavasti, ja toistamistaajuutta lyhennetään. Toistamistaajuuden lyhentäminen on yksisuuntainen muutos, jota ei voida peruttaa takaisin alkuperäiseen arvoon tuhoamatta algoritmin tuottamien yksilöiden sisältämää informaatiota. Toistamistaajuuden lyhentäminen myös kasvattaa yksilöiden generointiin vaadittua laskenta-aikaa, minkä takia taajuutta ei kannata lyhentää liian aikaisiin.

5.2 Tason paloittelu optimointia varten

Koska algoritmin hakualue kasvaa eksponentiaalisesti tason pituuteen verrattuna, haetun ratkaisun aikajakson lyhentäminen olisi erittäin hyödyllistä algoritmin suorituskyvyn kannalta. Tämän takia eräs mahdollinen suhteellisen helposti lisättävissä oleva optimointi olisi pelitason paloittelu erillisiin pienempiin osiin, jotka ovat ratkaistavissa nopeammin kuin yksi pidempi jatkuva taso.

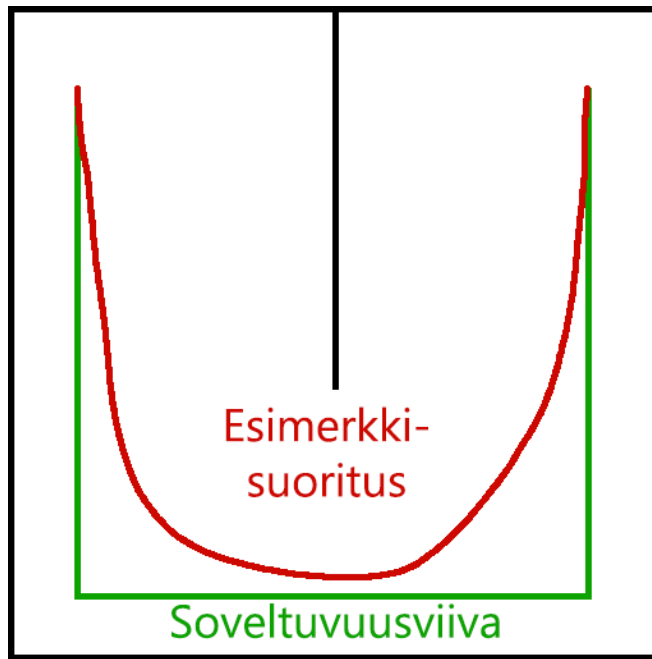
Pelitason paloittelun voisi toteuttaa hyödyntäen jo olemassa olevaa soveltuvuusarvon mittaamiseen käytettyä viivaa, koska se muodostuu useammasta kahden pisteen välisestä osasta. Jokainen kahden pisteen välinen osa voisi olla oma ”taso” algoritmin näkökulmasta, joiden suoritukset yhdistetään toisiinsa.

Suorituksien saumatonta yhdistämistä varten pelaajahahmon tila tulisi tallentaa osioiden välillä. Yhden osion lopussa oleva pelaajahahmon tila tulisi olla seuraavan osion aloitustila. Pelaajahahmon tilan tallennettavia arvoja ovat esimerkiksi pelaajahahmon sijainti, nopeus ja orientaatio.

Tason paloittelu pienempiin osiin on teoriassa erittäin toimiva optimointimenetelmä, koska sen avulla voidaan siirtää osa hakualueen eksponentista sen kertoimeksi. Esimerkiksi jos kolmen sekunnin pituinen taso paloiteltaisiin kolmeen erilliseen sekunnin pituiseen osaan, käyttäen aiempaa kaavaa binääriarvojen hakualueen arviointiin, hakualue ennen paloittelua olisi $5^{\frac{1000 \cdot 3}{15 \cdot 20}}$ ja paloittelun jälkeinen osioiden hakualue olisi $3 * 5^{\frac{1000}{15 \cdot 20}}$.

Koska jokaisen osion aloitustila pohjautuu edellisen osion lopetustilaan, on helppo kuvitella tilanteita, joissa jokin osio on mahdotonta läpäistä huonon aloitustilan takia. Tason paloittelun implementaation suurimpana ongelmana onkin oletettavasti heuristiikka, jonka perusteella osio katsotaan olevan läpäisty hyväksyttävästi, ja seuraavaan osioon eteneminen sallitaan. Osion lopputilan täytyy olla tarpeeksi hyvä, jotta seuraavan osion läpäisy on mahdollista. Tämän takia osion ratkaisun soveltuvuusfunktion tulisi huomioida matkustetun etäisyyden lisäksi muun muassa pelaajan orientaatio, nopeus ja nopeuden suunta.

Pelaajahahmon etäisyyttä soveltuvuusviivaan ei myöskään kannata painottaa liian paljoa osioiden lopussa, koska jokainen osio on suora viiva ja niiden väliset kulmat ovat usein suuria. Tämän takia soveltuvuusviiva itse ei ole optimaalinen polku tason läpi, vaan sen on tarkoitus ohjata algoritmia oikeaan suuntaan. Tason läpi kulkeva polku on yleisesti ottaen parempi, jos pelaajahahmon kulkevan polun suunta muuttuu sulavammin ajan myötä ja jos kuljettu etäisyys on mahdollisimman lyhyt, kuten kuvassa 15.



Kuva 15. Ylhäältä päin kuvattu 2D-pelitaso, jossa näkyy soveltuvuusviiva (vihreä) ja esimerkkisuorituksen kulkema polku tason läpi (punainen).

6 Yhteenveto

Työssä tutkittiin tavanomaisen geneettisen algoritmin sovellettavuutta liikkumiseen Source-pelimoottorin Surf-pelimuodossa. Geneettinen algoritmi on ollut erittäin hyvin dokumentoitu käsite tietoteknisessä kirjallisuudessa jo vuosikymmeniä, ja olemassa oleva teoria on sovellettavissa myös 3D-pelimaailmassa navigointiin. Tyypillinen geneettinen algoritmi sopeutuu tehtävään hyvin, eikä liikkeen tuottaminen peleissä välttämättä vaadi esoteerisempia ratkaisuja. Jo olemassa oleva teoreettinen pohja tarjoaa myös lupaavia jatkokehitysideoita sovelluksen optimointia varten.

Työn käytännön toteutus ja testaus onnistuivat ilman suuria ongelmia, ja jo tähän mennessä tuotetut tulokset vaikuttavat lupaavilta sovelluksen jatkokehityksen kannalta. Työhön liittyvän kirjallisuuden tutkiminen ei myöskään tuottanut ongelmia. Työn prosessin sujuvuuden voidaankin suurilta osin katsoa johtuvan laajasta kirjallisesta pohjasta geneettisiin algoritmeihin liittyen sekä käytetyn SourceMod-ohjelmointirajapinnan hyvästä dokumentaatiosta ja aiemmasta käytännön kokemuksesta.

Työn lopputulokset ovat laadultaan hyviä ja julkisesti saatavilla. Sovelluksen suorituksesta nauhoitetulla videolla näkyvä liike on sulavaa ja pelitasojen suoritukset ovat suurimmilta osin tavanomaisia. Työn toteutuksen aikana tuotettu ohjelmistokoodi on julkaistu GPL-3.0-lisenssin alhaisena avoimena lähdekoodina.

Geneettisen algoritmin soveltaminen monimutkaisen liikesarjan löytämiseen ja suorittamiseen 3D-peleissä on mahdollista nykyaikaisen laitteiston tarjoaman laskentatehon myötä. Geneettinen algoritmi voi aloittaa satunnaisesti tuotetulla populaatiolla ja kykenee ajan myötä löytämään sarjan liikekomentoja läpäistäkseen Surf-pelitasoja ilman tietämystä pelin mekaniikoista. Käytännön osuudessa esitetyt yksinkertaiset lyhyet tasot toimivat todisteena sovelluksen toimivuudesta.

Jatkokehitys-osioissa esitetyiden optimointimenetelmien implementoinnin myötä sovelluksen pitäisi teoreettisesti pystyä läpäisemään monimutkaisempia ja pidempiä tasoja, mutta näihin vaadittu laskennallinen aika ja käytännöllisyys jää nähtäväksi.

Lähteet

- 1 Baluja S, Caruana R. Removing the Genetics from the Standard Genetic Algorithm. Carnegie Mellon University; 1995. Saatavilla: https://www.ri.cmu.edu/pub_files/pub2/baluja_shumeet_1995_1/baluja_shumeet_1995_1.pdf. Haettu 04.11.2020.
- 2 Reeves CR, Rowe JE. Genetic Algorithms - Principles and Perspectives : A Guide to GA Theory. Secaucus: Kluwer Academic Publishers; 2002.
- 3 Bäck T. Evolutionary Algorithms in Theory and Practice : Evolution Strategies, Evolutionary Programming, Genetic Algorithms. New York: Oxford University Press, Incorporated; 1996.
- 4 Conceptually. Determinism - Explanation and examples. Saatavilla: <https://conceptually.org/concepts/determinism>. Haettu 27.10.2020.
- 5 Valve Developer Community. Source. Saatavilla: <https://developer.valvesoftware.com/wiki/Source>. Haettu 29.11.2019.
- 6 Valve Corporation. Counter-Strike: Source. Saatavilla: https://store.steampowered.com/app/240/CounterStrike_Source. Haettu 01.11.2020.
- 7 Valve Corporation. Half-Life 2. Saatavilla: https://store.steampowered.com/app/220/HalfLife_2. Haettu 01.11.2020.
- 8 Valve Corporation. Team Fortress 2. Saatavilla: https://store.steampowered.com/app/440/Team_Fortress_2. Haettu 01.11.2020.
- 9 Valve Corporation. Counter-Strike: Global Offensive. Saatavilla: https://store.steampowered.com/app/730/CounterStrike_Global_Offensive. Haettu 01.11.2020.
- 10 Wright ST. The unlikely origin of Counter-Strike surfing. Eurogamer. Saatavilla: <https://www.eurogamer.net/articles/2019-02-15-meet-the-guy-who-accidentally-invented-surf-maps-in-counter-strike>. Haettu 02.11.2020.
- 11 Adrian Biagioli. Bunnyhopping from the Programmer's Perspective. Saatavilla: <https://flafla2.github.io/2015/02/14/bunnyhop.html>. Haettu 01.11.2020.

- 12 Valve Developer Community. Goldsource. Saatavilla: <https://developer.valvesoftware.com/wiki/Goldsource>. Haettu 03.11.2020.
- 13 SourceMod Dev Team. SourceMod. Saatavilla: <https://www.sourcemod.net/about.php>. Haettu 03.11.2020.
- 14 Kim IY, de Weck O. Variable Chromosome Length Genetic Algorithm for Structural Topology Design Optimization. Cambridge: Massachusetts Institute of Technology; 2004. Saatavilla: http://web.mit.edu/deweck/www/PDF_archive/3%20Refereed%20Conference/3_29_AIAA_2004_1911.pdf. Haettu 01.11.2020.

Liitteet

- 1 Sovelluksen lähdekoodi. <https://github.com/laurirasanen/ga-input>
- 2 Video geneettisen algoritmin suorituksesta *surf_beginner*-pelikentän tasoissa 1-4.
https://laurirasanen.github.io/pub/ga_surf_beginner.mp4