CHALMERS

Genetiska algoritmer i realtid för spelutveckling

Kandidatarbete inom data- och informationsteknik

DAVID GRANKVIST
ROBIN HAMMARÄNG
FANNY MALMEK
JESPER PERSSON
OLLE WERME

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2014

Abstract

This paper discuss genetic algorithms and their applicability in game development. The possibility to generate a diversified response from a game depending on the player's choices or strategies is evaluated, as an alternative to programming each possible event beforehand.

The paper describes the implementation of a prototype, which uses genetic algorithms in real-time. The prototype is modelled after a simple game model to serve as an evaluation platform for the algorithms.

Based on analyzed data from the prototype conclusions are drawn about the impact of the genetic algorithm on the game's behavior, user experience and their usefulness in the area.

Sammanfattning

Detta arbete behandlar genetiska algoritmer och deras applicerbarhet inom spelutveckling. Arbetet utvärderar om det med hjälp av dessa algoritmer går att evolvera en varierande respons från spelet beroende på spelarens val eller strategier. Detta som alternativ till att programmera statiska händelseförgreningar på förhand.

Arbetet redogör kring implementationen av en prototyp, där genetiska algoritmer används i realtid. Prototypen är utformad efter en simpel spelmodell och fungerar som en plattform för utvärdering av algoritmerna.

Utifrån analyserad data från prototypen har slutsatser om den genetiska algoritmens påverkan på spelets utveckling, användarupplevelse och deras användbarhet inom området kunnat dras.

Innehållsförteckning

Kapitel 1

Inledning

Ett framträdande problem inom spelutveckling är svårigheten med att åstadkomma en varierande händelseutveckling. Hur ett spel förgrenar sig beror på många olika variabler och i de fall där spelarens val och spelstil påverkar hur spelet beter sig behöver varje möjlig förgrening programmeras på förhand. Oavsett hur mycket tid eller pengar som finns tillgängliga kan endast ett ändligt antal förgreningar implementeras. Detta resulterar i linjära spel där två olika spelare ofta möts av samma händelseförlopp.

Genom att applicera genetiska algoritmer skulle delar av ett spels utvecklingsförlopp kunna genereras utifrån spelarens val och strategier. Detta skulle leda till en mer dynamisk spelupplevelse, där olika spelarbeteenden besvaras av varierande respons från spelet utan att dessa förgreningar programmeras in på förhand.

1.1 Syfte

Detta arbete har som syfte att utforska hur genetiska algoritmer kan appliceras inom spelutveckling och köras under spelets gång för att skapa en mer dynamisk spelupplevelse med varierande händelseutveckling. Detta genomförs genom implementering och analys av en spelprototyp. Slutsatser dras utifrån om och hur väl de genetiska algoritmerna fungerar samt vilken effekt det har på spelupplevelsen.

1.2 Utmaningar

Genetiska algoritmer är till viss del styrda av slumpen och det är därför svårt att garantera specifika resultat. För att få stabila värden krävs

vanligtvis många generationer, vilket kan ta lång tid. För körning i realtid är det därför viktigt att algoritmen anpassas för detta så att resultat kan uppnås snabbare. Ytterligare en utmaning är att få algoritmerna att ge en stabil utveckling på få generationer, utan att begränsa antalet möjliga resultat för mycket.

Kapitel 2

Genetiska algoritmer

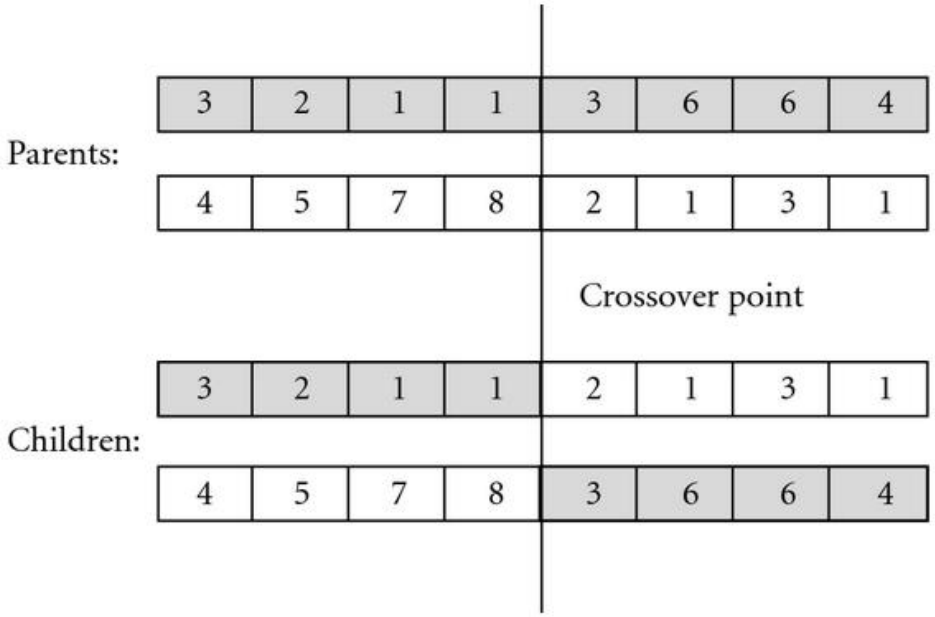
Genetiska algoritmer är stokastiska processer som iterativt löser eller estimerar lösningar till optimerings- och sökproblem (Wahde, M.W., 2008). Algoritmerna använder sig av tekniker som är inspirerade av naturligt urval och evolution. Principerna från teorin om det naturliga urvalet visar sig exempelvis genom hur tekniken identifierar viktiga karaktäristiska drag som krävs för att lösa en specifik uppgift och överför dessa drag till nästa iteration av lösningen. Dessa egenskaper är vad som gör algoritmerna användbara och anpassningsbara för olika typer av problem.

2.1 Introduktion

Den centrala beståndsdelen i de genetiska algoritmerna är en population av individer, i vilken varje individ representerar en lösning på optimerings- eller sökproblemet i fråga. Algoritmerna simulerar sedan utvecklingen av populationen, generation efter generation. Processen är iterativ och i varje generation utvärderas varje individ efter en målfunktion. Invididerna tilldelas en poäng baserat på utvärderingen (Wahde, M.W., 2008). Den första populationen som skapas är ofta stokastisk medan de följande generationerna är framtagna baserade på föregående generation. De individer som bäst löser problemet tilldelas en hög poäng och har därmed störst chans att bli en del av kommande generationer. Utvalda individer genomgår korsning och mutation, två metoder som används för att ta fram de nya individer som utgör nästa population.

2.1.1 Korsning

Korsning kombinerar två individer för att skapa två nya (Wahde, M.W., 2008). Detta görs genom att kapa individerna vid en eller flera godtyckliga punkter, och sedan bygga nya individer av kombinerade delar enligt figur 2.1. Korsning är ett effektivt verktyg för att sprida en individs egenskaper i populationen.



Figur 2.1 - Två föräldrar delas och korsas till två nya individer redo att användas i nästa generation (del Notario, C.B.P.N., Baert, R.B., and D'Hondt, M.H., 2012).

2.1.2 Mutation

Mutation är ett sätt för de genetiska algoritmerna att hitta mångfald bland lösningarna (Wahde, M.W., 2008). Genom att slumpmässigt mutera små delar av en individs egenskaper kan nya aspekter av en lösning hittas. Vanligtvis ger inte mutation någon framgång på ett kortsiktigt perspektiv, men det kan över längre perioder hjälpa till att hitta lösningar som inte tidigare hanterats. Sannolikheten för mutation är en av de parametrar som vanligtvis anpassas mellan olika problem.

2.1.3 Representation av individer

Hur individer representeras i en genetisk algoritm beror på sammanhanget där den genetiska algoritmer appliceras (Wahde, M.W., 2008). I den klassiska beskrivningen av genetiska algoritmer representeras varje individ av en mängd ettor och nollor. Denna representation av individer kallas för binär kodning.

Det finns dock nackdelar med binär kodning; ett konkret exempel är den mycket effektiva mutation den orsakar (Wahde, M.W., 2008). I binär kodning implementeras mutation genom att invertera enskilda bitar. Om till exempel den mest signifikanta biten i en binär individ muteras kan det orsaka en drastisk förändring hos individens egenskaper, vilket inte är önskvärt om mutation i sammanhanget har syftet att göra små förändringar. Denna typ av stabilitetsproblem kan undvikas genom att koda individer med graykod eller med reella tal (Wahde, M.W., 2008).

Graykod är ett binärt räknesystem sådant att givet ett tal når man nästa tal i talföljden genom att manipulera en enskild bit. Kodas en individ på detta sätt ökar därmed den genetiska algoritmens stabilitetsfaktor i och med att manipulering av enskilda bitar bara gör mindre förändringar. I fallet av att koda individer med reella tal undviks konceptet med binära tal och signifikanta bitar. Reell kodning har dessutom fördelen att om optimeringsproblemet i fråga behandlar en samling reella tal krävs ingen överflödig konvertering mellan talbaser.

2.1.4 Elitism

Elitism är ett koncept som är framtaget för att få märkbara resultat snabbare, genom att se till att den eller de bästa individerna från en generation alltid överförs oförändrade till nästa (Wahde, M.W., 2008). Utan elitism finns en risk att även de dittills bästa funna lösningarna går förlorade mellan generationerna, då det inte finns någon garanti för att den bästa individen ska väljas. Ett annat problem som kan undvikas med hjälp av elitism är att mutation av den bästa individen, eller en korsning

med andra individer, förstör de egenskaper som gjort individen framgångsrik.

2.1.5 Målfunktion

Målfunktionen är central för att de genetiska algoritmerna ska fungera bra och är en av de delar som användaren själv måste specifiera inför varje problem. Målfunktionen ska tydligt reflektera resultatet av en lösning (Haupt, R.L.H., Haupt, S.E.H., 2004). Exempelvis kan de genetiska algoritmerna användas för att hitta kortaste sträckan mellan en eller flera punkter. I detta fall kan inversen för summan av alla sträckor lämpligen användas som målfunktion.

2.2 Styrkor

En styrka med genetiska algoritmer är att de kan appliceras på många olika problem av varierande komplexitet, utan att algoritmen behöver genomgå några omfattande förändringar (Haupt, R.L.H., Haupt, S.E.H., 2004). Det som framför allt skiljer sig mellan olika applikationer är målfunktion, storlek på population, mutationschans och vilken grad av elitism som skall användas.

2.2.1 Anpassningsbarhet

Algoritmerna kan anpassas och hitta lösningar på allt från enkla ordgissningslekar till klassiska ingenjörsproblem som Traveling Salesman Problem (Haupt, R.L.H., Haupt, S.E.H., 2004). De kan även appliceras på ämnen inom datavetenskap som exempelvis avkodning av hemliga meddelanden eller optimering av artificiella neurala nätverk. Även mer traditionella matematiska problem som lösning av ickelinjära partiella differentialekvationer av högre ordning kan åstadkommas med hjälp av algoritmerna.

Då algoritmerna under 2000-talet har fått mer uppmärksamhet än tidigare innebär det att det ständigt upptäcks nya appliceringsområden. Att

användningsområdena för de genetiska algoritmerna är så många beror till stor del på att algoritmerna hanterar både om problemet behandlar kontinuerliga eller diskreta variabler, om ett matematiskt problem saknar derivata, hur många variabler som skall optimeras samt om datan som algoritmerna analyserar är numeriskt genererad data, experimentell data eller kommer från andra analytiska funktioner (Haupt, R.L.H., Haupt, S.E.H., 2004).

2.2.2 Stor sökrymd

Att de genetiska algoritmerna använder sig av en iterativ process innebär att inte bara en lösning, utan ett set, hittas (Haupt, R.L.H., Haupt, S.E.H., 2004). Användaren skriver sin egen målfunktion och bestämmer därmed själv när en tillräckligt bra lösning har hittats, men presenteras även med alla andra lösningar som funnits. Skulle användaren inte vara nöjd med den bästa lösningen som hittats, kan man även gå tillbaka och utvärdera övriga lösningar. Lösningarna är dessutom delvis stokastiska och på så sätt kan algoritmerna hitta flera lösningar som ger likvärdiga resultat men som skiljer sig helt från varandra (Wahde, M.W., 2008).

2.3 Svagheter

En svaghet med genetiska algoritmer är att de är approximativa och inte nödvändigtvis garanterar den bästa lösningen. Målfunktionen skrivs av användaren själv, och vid problem som inte är väldefinierade är det svårt att formulera en målfunktion som ger den bästa lösningen (Haupt, R.L.H., Haupt, S.E.H., 2004). På grund av detta hittas ofta inte den bästa lösningen, utan bara ett set av bra lösningar.

2.3.1 Lokalt optimum

Ett problem kopplat till målfunktionen är att användaren bestämmer när ett problem är löst (Haupt, R.L.H., Haupt, S.E.H., 2004). Då den iterativa processen inte är ändlig behöver ett beslut tas angående när resultaten är tillräckligt bra för att accepteras som lösning. För ett svårformulerat

problem kan detta exempelvis vara att resultatet inte blivit bättre på ett bestämt antal generationer. Detta kan resultera i att algoritmen hittar ett lokalt optimum istället för ett globalt sådant. Med andra ord kan det finnas bättre lösingar, men algoritmen väljer fel väg att gå och hittar därmed inte andra, potentiellt bättre, lösningar.

2.3.2 Justering av parametrar

Genetiska algoritmer bygger på ett antal parametrar som användaren själv måste ange och justera vid olika problemformuleringar. Storlek på populationerna, hur fort mutation ska ske, hur individerna tar del i korsnings-processen och elitism är några av de aspekter användaren måste ta hänsyn till för att få bra resultat. Olika problem kan ge avsevärt skilda resultat beroende på vilka val av parametrar som gjorts. Det finns inte heller någon konfiguration som är konsekvent överlägsen för en större antal problem (Wahde, M.W., 2008).

2.3.3 Prestation relativt analytiska metoder

Vid mer klassiska tillämpningar, inom till exempel matematik och fysik, finns det dessutom olika analytiska metoder som kan lösa problemen. De analytiska metoderna hittar i de flesta fall lösningar snabbare som dessutom är bättre, relativt de genetiska algoritmerna (Haupt, R.L.H., Haupt, S.E.H., 2004).

2.4 Realtidsapplicering

En del problem kan behöva en körning för att hämta all data som behövs. Till exempel när en mänsklig bedömning krävs, eller när individers uppförande under en körning eller simulering ligger till grund för bedömningen. En sådan realtidsanvändning av en genetisk algoritm innebär alltså att algoritmen arbetas med i realtid istället för att den itererar klart och resultaten hämtas i efterhand.

2.4.1 Interaktion och simulering

Interactive evolutionary computation (IEC) är namnet på problemområdet där mänsklig interaktion krävs för att bedöma lösningar (Wahde, M.W., 2008). När evolutionära algoritmer används för att generera estetik är det ofta exempel på en IEC.

Musikgeneratorn GenJam, som ska simulera en improviserande musiker, kräver att en människa lyssnar och avgör hur bra det låter (Biles, J.A.B., 1994). Stycken som genereras av GenJam utvärderas kontinuerligt av en mänsklig bedömare och data från denna utvärdering används sedan för att genera en ny generation av stycken. På detta sätt kan riktiga musikstycken skapas i samarbetet mellan algoritm och mentor.

2.4.2 Svårigheter

Eftersom evolution och evolutionära algoritmer i allmänhet är långsamma processer kan realtidsappliceringar vara problematiska (Wahde, M.W., 2008). När man i andra fall har en algoritm som körs i flera tusen generationer utan avbrott, måste man här manuellt köra varje generation och utvärdera denna. Detta leder till att man sällan får den stabilitet och säkerhet som flera iterationer ger.

Själva utvärderingen och målfunktionen kan i många fall vara diffus då det är en människa och inte en algoritm som avgör vad som är nära målet och inte. Människor är inte lika konsekventa som datorer vilket kan medföra att individer som ansågs vara framgångsrika i en tidigare generation inte nödväntigtvis anses vara det i en senare generation.

2.5 Spelutveckling

Genetiska algoritmer har ännu inte använts mycket inom spelbranschen, även om de uppmärksammas mer och mer för sin potential inom bland annat artificiell intelligens (Charles, D.C., Fyfe, C.F., Livingstone, D.L.,

McGlinchey, S.M., 2008). När genetiska algoritmer tidigare har applicerats i området har de framförallt använts som ett verktyg för att balansera och optimera slutprodukten. Av denna anledning är algoritmerna sällan kvar i den slutgiltiga produkten.

Att generera fram beteenden för ett spels olika enheter kan ha många fördelar. Exempelvis är det inte lika tidskrävande som att utveckla hur komplicerad artificiell intelligens tar beslut. Genetiska algoritmer kan dessutom hitta användbara beteenden som utvecklare förbiser (Lucas, S.M.L., Kendall, G.K., 2006). Genererade beteenden kan på detta sätt leda till intressanta spel med oväntad och intressant strategisk respons.

Genetiska algoritmer kan användas för att balansera spel. Genom att implementera enheter eller karaktärer med genetiska algoritmer kan exempelvis olika balansförstörande taktiker och oönskade genvägar i spel upptäckas (Denzinger, J.D., Loose, K.L., Gates, D.G and Buchanan, J.B., 2005).

Kapitel 3

Metod

Utöver planeringsstadiet där strukturen för arbetet lades upp, kan arbetsgången naturligt delas in i tre tydliga i faser beskrivna i följande avsnitt. Avsnittet behandlar också de metoder och verktyg som använts under projektets gång.

3.1 Förstudie

Förstudiens huvudsakliga syfte var att förbereda implementeringsstadiet genom att fördjupa kunskapen om genetiska algoritmer och hur dessa tidigare använts i området. Genom att analysera andra implementeringar förtydligades också bilden av vilken slags prototyp som skulle utvecklas. Resultatet för denna del av studien beskrivs mer genomgående i föregående kapitel.

3.1.1 Intervju

Efter att upplägget för prototypen fastställts genomfördes en intervju med Mattias Wahde, professor inom tillämpad mekanik på Chalmers tekniska högskola, som undervisar i kursen Stochastic Optimization Algorithms. Syftet med mötet var att redovisa prototypidén samt diskutera eventuella svårigheter för att i möjlig utsträckning förhindra tidiga felsteg under implementeringen.

3.1.2 Språk och ramverk

Även mer konkreta aspekter som programmeringsspråk och ramverk undersöktes. För att maximera effektivitet och bibehålla fokus på algoritmimplementering behövde verktygen vara väl anpassade för uppgiften. De mest centrala aspekter som iakttogs var:

- Prestanda
- Dokumentation
- Omfång

Den typ av spelimplementation som planerades krävde en omfattande kodbas. Det var därför av stor vikt att hitta en platform där basfunktionalitet för exempelvis hantering av grafik, I/O och enhetskollision fanns färdig. Kvaliteten på dokumentation och mängden exempelkod var också viktiga faktorer då implementeringen behövde komma igång snabbt.

3.1.3 Phaser

Efter en analys av flera kända spelramverk valdes det JavaScript-baserade *Phaser*. Phaser är ett funktionsmässigt mycket omfattande spelramverk med fokus på webbplatformen (Davey, R.D, 2014). Ramverket besitter alla de egenskaper som söktes utan att ålägga någon bestämd klasstruktur för utvecklare. Detta möjliggjorde snabb implementering av även mer invecklad spellogik samt full frihet i hur kod skulle struktureras.

3.2 Implementation av spelmodell och algoritmmodul

Målet med implementationsdelen av arbetet var att snabbt få igång en fungerande prototyp som kunde agera simuleringsmotor för spelmodellen som konkretiserats under förstudiefasen.

3.2.1 Basfunktionalitet för prototyp

Modellen för spelet utformades för att på ett så framgångsrikt och enkelt sätt som möjligt kunna simulera evolutionslikande utvecklingsmönster för datorstyrda enheter. Mer om spelkonceptet och utvecklingsprocessen redovisas i kapitel 4.

Under detta stadie utvecklades också grunden för algoritmmodulen parallellt. Avsikten var att skapa en fristående modul som inte var beroende av spelkoden. Mycket fokus lades på prestandatester då detta var en ännu osäker och avgörande aspekt för hur det fortsatta arbetet skulle komma att se ut.

3.2.2 Utvecklingsmetodik

För att maximera tidseffektiviten användes utvecklingsmetodiken *Scrum* under projektet. Metodiken har starkt fokus på anpassning till produktförändring och samarbete inom utvecklingsgruppen (Schwaber, K.S., Sutherland, J.S., 2013) vilket förutspåddes passa projektet bra.

3.3 Analys och balansering av prototyp

Efter att algoritmmodulen kopplats samman med resten av kodbasen övergick arbetet till en iterativ process av att justera spelmodellen och algoritmerna för att skapa en välbalanserad prototyp. En välbalanserad prototyp innebär att de olika fiendetyperna presterar bra mot vissa spelstrategier och sämre mot andra. Arbetsgången betecknades av löpande utveckling, praktisk testning och diskussioner.

Från ett spelupplevelseperspektiv var målet att skapa en produkt där användaren märkte tydlig skillnad på hur spelet anpassat sig beroende på spelstil. Med ett evolverande motstånd skulle till exempel en konstant aggressiv spelare kunna mötas av stark defensiv respons. För att förtydliga

detta anpassande beteende används en förenklad modell där motståndsenheterna följer ett av flera bestämda beteendemönster.

Balanseringsprocessen bestod i huvudsak av två delar: Experimentering med förmågor och attribut för motståndarenheter samt justering av hur algorimtmodulen efter varje spelomgång utvärderar dessa.

Kapitel 4

Prototyp

Följande avsnitt beskriver utvecklingen av en prototyp av en enklare spelmodell vars slutgiltiga version var tänkt att fungera som ett koncept för hur genetiska algoritmer med fördel kan användas i realtid i spel. Spelidén finns beskriven i Appendix A.

4.1 Koncept

Den idé som ursprungligen lade grunden till detta arbete var tanken på ett spel där motståndet med hjälp av genetiska algoritmer utvecklas och anpassar sig för att bättre bemöta spelaren. Idén utvecklades under förstudien till ett mer konkret spelkoncept med fokus på att tydligt simulera ett stabilt evolutionärt beteende.

Spelmodellen består av ett upprepande överlevnadsmoment förklarat i figur 4.1. I den stängda miljön styr spelaren en karaktär som kontinuerligt attackeras av en grupp fiendeenheter. Som spelare kan man välja att vara aggressiv och attackera fienderna eller överleva genom att spela defensivt och hålla sig undan. När alla fiendeenheter blir besegrade eller tiden för omgången tar slut, avslutas den. Detta moment upprepas ändlöst och mellan varje omgång genereras en ny fiendevåg baserat på resulten från den föregående. För att vidhålla en balans gentemot det ständigt evolverande motståndet ges efter varje avklarad omgång ett antal poäng till spelaren som kan distribueras över olika attribut.

Figur 4.1 - Prototypen som utvecklas är av ett enkelt överlevnadskoncept där spelaren styr en karaktär bland ständigt attackerande fiendeenheter. Spelet är omgångsbaserat och nya evolverade fiendepopulationer genereras i början av varje runda.

4.2 Modell för effektiv evolution

För att förtydliga det evolverande mönstret prototypen har för avsikt att visa behövde modellen förenklas för att effektivt kunna bemöta olika spelarbeteenden. Lösningen blev att utveckla olika typer av fiendeenheter med unika förmågor och beteenden framtagna för att vara särskilt effektiva mot ett specifikt spelarbeteende.

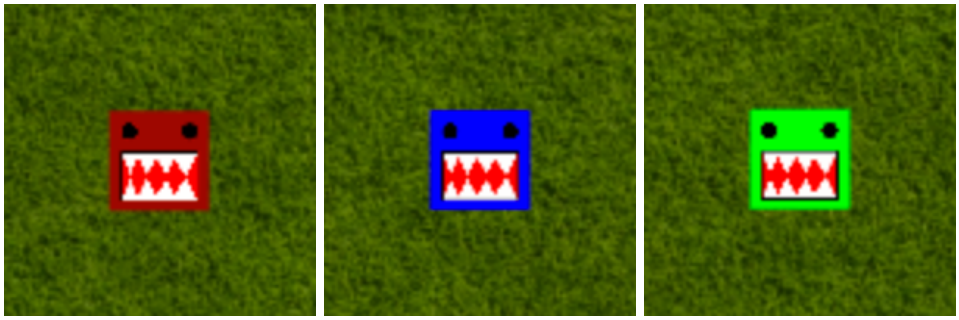
4.2.1 Begränsad spelarkaraktär

Spelmodellen begränsar avsiktligt spelares egenskaper för att främja tre huvudsakliga beteenden; antingen går spelaren i närstrid med fiendeenheterna, skjuter ner dem på avstånd eller väljer att helt försöka

undvika dem. De tre beteendena kan i realiteten kombineras men för enkelhetens skull antas spelaren i huvudsak hålla sig till en av dessa strategier.

4.2.2. Anpassade fiendeenheter

För varje spelarbeteende finns en motsvarande typ av fiendeenhet. Möts denna fiendeenhet av spelarbeteendet i fråga skall denna presterar bättre än andra typer. Följande tre fiendetyper, illustrerade i figur 4.2, förutspåddes vardera besvara en av spelarens strategier. Strategierna och vilken fiendetyp som spelaren väntas möta beskrivs i tabell 4.3.



Figur 4.2 - Olika fiendetyper skapas för att öka möjligheten att matcha olika spelarbeteenden. För tydlighetens skull representeras varje fiendetyp av en färg; fiender för närstrid är röda, de skjutande är blåa och de snabba är gröna.

Sett från ett evolutionärt perspektiv skulle till exempel en undvikande spelstil resultera i att de snabba fiendeenheterna fick ett tydligt övertag och hade större chans att överleva i kommande generationer. Med denna förenklade modell troddes spelet kunna tydliggöra och snabba på de evolutionära effekterna av en genetisk algoritm.

Tabell 4.3 - Spelarbeteenden och de fiendeenheter dessa effektivt kontras av

Spelarbeteende	Kontras av fiendeenhet
Aggressiv (Närstrid)	**Skjutande** En enhet med förmågan att skjuta. Enheten tar inga initiativ till att ta sig närmre spelaren så länge den har en ren skottvinkel och backar när spelaren närmar sig.
Avvaktande (Skjutande)	**Stark** En stark enhet som utöver sin närstridskapabilitet kan teleportera sig själv kortare sträckor över spelplanen. Enheten jagar spelaren och teleporterar sig när möjligt in precis bredvid denna.
Undvikande (Flyende)	**Snabb** En snabb enhet som är rörligare än övriga typer. Enheten kan även göra sig själv odödlig i korta perioder när den tar emot skada. Enheten rör sig också snabbare när den är odödlig.

4.3 Attribut och fiendetyper

Både spelarkaraktär och fiendeenheter är grundade i ett antal gemensamma attribut. Dessa är centrala då de används i algoritmmodulen för generera nya enheter samt ligger till grund för alla dess egenskaper. Attributen är styrka, intelligens och rörlighet. Dessa tre grundattribut avgör nivån på olika egenskaper för varje enhet. Styrka avgör hur mycket skada en enhet kan göra per attack samt hur mycket den kan ta emot innan den stupar. Rörlighet avgör hur snabbt en enhet kan röra sig samt hur uthållig den är. Intelligens avgör slutligen hur bra en enhet siktar med skjutvapen samt i hur hög frekvens den kan avfyra projektiler.

Attributen avgör vilken fiendetyp som genereras, alltså vilka förmågor och beteenden varje enhet blir tilldelad. Det som avgör vilken typ en fiende tilldelas är vilket attribut som har högst värde, enligt tabell 4.4.

Tabell 4.4 - Fiendetyp och dess huvudattribut

Attribut	Fiendetyp
Styrka	Teleporterande närstridenhet
Intelligens	Projektilavfyrande avståndsenhet
Rörlighet	Jagande snabb enhet

4.4 Implementation av den genetiska algoritmen

Den genetiska algoritmens uppgift i prototypen är att efter varje spelomgång utvärdera hur väl fienderna klarat sin uppgift och generera nya fiendepopulationer anpassade för att möta spelarbeteendet. Implementationen av algoritmen i sin helhet kan läsas i appendix B.

4.4.1 Kodning med reella tal

I prototypens genetiska algoritm används reella tal för att representera individers gener. Anledningen till att kodning med reella tal används snarare än binär kodning är att det ger en tydligare datarepresentation av prototypens enheter när de översätts till individer i den genetiska algoritmens population. Varje individ har tre gener och varje gen representerar ett av de tre grundattributen, vilket intuitivt beskriver den översatta enheten. Konverteringen mellan algoritmens individer och spelets enheter kan läsas i Appendix C.

4.4.2 Målfunktion

Optimeringsproblemet som prototypen ska lösa är att för hela populationen hitta de fördelningar av grundattribut hos fienderna som på bästa sätt bemöter spelarens strategi. Huruvida en fiende närmar sig en optimal fördelning av grundattributen kan beskrivas på många sätt beroende på önskad komplexitet.

I denna prototyp valdes en enkel modell för att beskriva fiendernas mål. Oavsätt spelarens strategi är fiendernas mål att döda spelaren. Modellen som används för att beskriva vilka enheter som lyckats bäst med detta är baserad på hur mycket skada de lyckats göra på spelaren. De fiender som har orsakat mycket skada tilldelas en hög poäng från målfunktionen och blir därmed framgångsrika i populationen.

4.4.3 Sannolikhet för mutation

För att hela tiden tvinga den genetiska algoritmen att hitta nya sätt att utveckla fienderna finns det hos varje individ, normalt sett, en liten chans att en eller flera gener muteras. Under implementationen av prototypen märktes mutationen sällan av, då sannolikheten för mutation sattes till 2.5% per gen. Eftersom mutationen appliceras per gen får det därmed oftast väldigt låga utslag på individen som helhet. De gånger hela populationen misslyckas med att lösa sin uppgift ökades chansen för mutation med faktorn 20, för att introducera nya egenskaper hos populationen.

4.4.3 Prestanda i realtid

Prototypen är designad för att med tillfredställande exekveringstid köra en genetisk algoritm i realtid. Som beskrivet i den tekniska bakgrunden finns det vanligtvis en viss interaktion mellan algoritm och användare i realtidsappliceringar. Detta gäller även i denna prototyp, där fiendernas framgång beror på spelstilen.

Under varje spelrunda samlas statistik om varje individuell fiende som sedan används för att tilldela den motsvarande individen i algoritmen en poäng från målfunktionen. I och med att denna interaktion krävs för att ge en representativ poäng till individer, genereras bara en ny generation av individer mellan spelrundor.

Ur ett prestandaperspektiv är det av stor fördel att bara generera en generation åt gången, då antalet generationer är avgörande för en genetisk algoritms exekveringstid. Detta tillvägasångsätt ökar dock risken för inkonsekventa och instabila resultat eftersom få generationer utvärderas.

4.5 Simuleringsläge för analys och balansering

I analys- och balanseringssyfte implementerades även en simuleringsmodul som en del av prototypen. Modulen fungerar som spelläget med undantaget att det inte finns någon spelarstyrd karaktär, istället genereras varje omgång två populationer fiendeenheter som slåss mot varandra.

Då modellen för spelarkaraktären utgörs av en förenklad kombination av de olika egenskaper fiendeenheterna besitter, förutspåddes ett simuleringsläge av denna typ kunna påvisa likande fenomen som i det spelarstyrda spelläget; en aggressiv närstridsenhet borde exempelvis i genomsnitt inte överleva om den ställs mot en avvaktande projektilavfyrande enhet och tvärtom. Med denna simuleringsmodul skulle då obalans i hypotesen om fiendetyper och strategier snabbt kunna upptäckas och justeras.

Kapitel 5

Resultat

Följande kapitel presenterar resultat från analytiska testkörningar av den färdiga prototypen. Avsnittet redovisar hur implementeringen av den genetiska algoritmen har fungerat i praktiken genom att analysera hur spelet besvarar olika strategier från spelaren. Insamlad data som ligger till grund för kommande diagram och slutsatser finns i Appendix D.

5.1 Utvärdering av respons på olika spelarbeteenden

Genom kontrollerade spelsessioner har olika spelarstrategiers bemötande kunnat analyseras separat från varandra och utvärderas. Varje strategi definierades tydligt på förhand och en spelare spelade sedan kontinuerligt sin strategi under ett bestämt antal nivåer innan spelet avslutades och en ny strategi testades. Attributfördelningen hos fiendeenheterna registrerades efter varje omgång och sammanställdes slutligen i ett diagram. På detta sätt har de olika strategierna kunnat jämföras med varandra och konkreta slutsatser kunnat dras. Spelarkaraktären gjordes under dessa tester odödlig för att förenkla processen.

De strategier som behandlats i utvärderingen följer det tidigare bestämda mönstret: aggressivt, avvaktande och undvikande. Nedan följer en mer genomgående beskrivning av resultaten för varje spelarstrategi.

5.1.1 Undvikande spelarstrategi

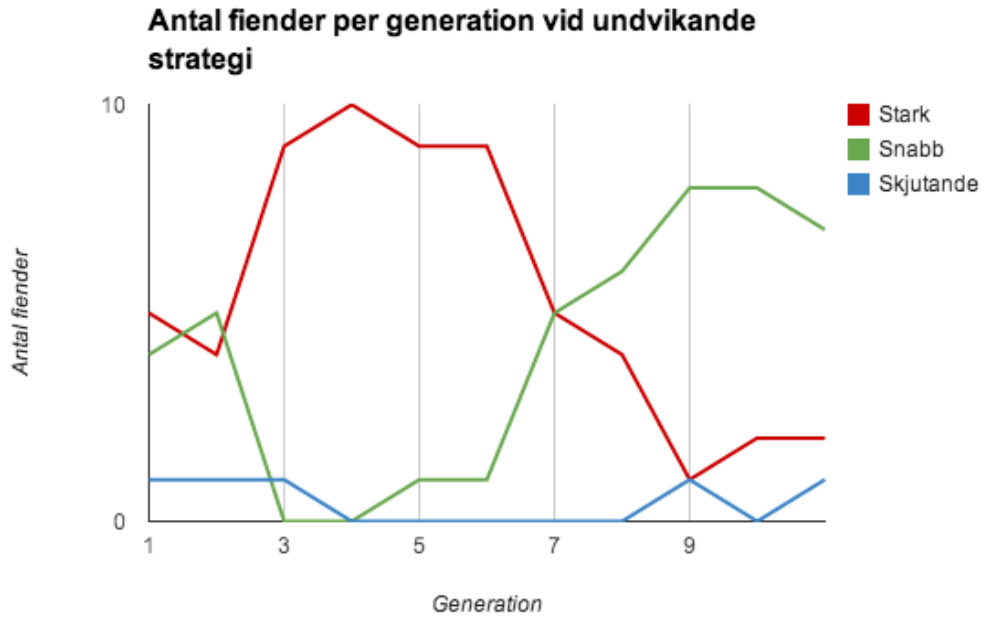
En undvikande spelare satsar sina attributpoäng på rörlighet och försöker hålla sig undan från fienderna tills tiden för den nuvarande spelomgången tar slut. Denna strategi har som förväntat visat sig fördelaktig för fiender

med höga värden på rörlighet, då den största utmaningen ligger i att komma ikapp spelaren.

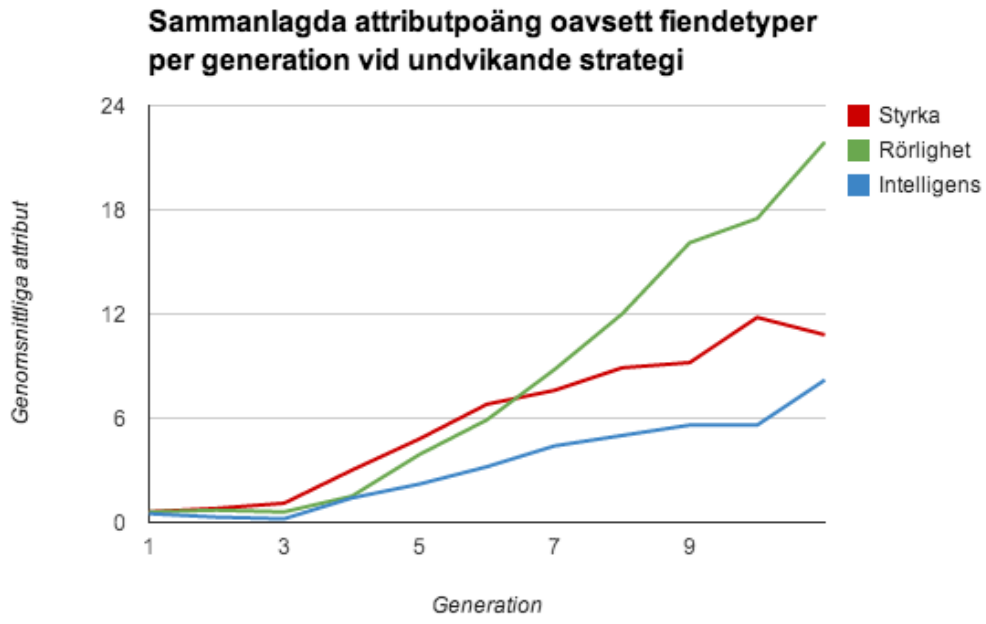
Då rörelsebaserade fiender använder sig av närstrid gynnas de också av att ha attributpoäng på styrka, vilket ökar deras förmåga att utdela skada. Detta resulterar i att de chanser enheten får att vålla skada på spelaren blir mer värdefulla. Då de skjutande fienderna ofta får in ett fåtal träffar per omgång finns även de närvarande, dock i mindre utsträckning än de övriga fiendetyperna.

Figur 5.1 visar distributionen av de olika fiendetyperna för varje generation. Under de första generationerna av spelomgången är de starka fienderna i stor majoritet men efter ett antal spelade nivåer skiftar detta och de snabba fienderna får istället övertag. Skjutnade fiender är under hela spelomgången frånvarande eller mycket få.

I figur 5.2 visas hur rörlighet har ett övertag över andra attribut, men också hur styrka är tydligt mer förekommande än intelligens. Att inte intelligens ligger lika högt kan bero på att så länge fienden inte är övervägande intelligent och kan skjuta, ger attributet inga större fördelar.



Figur 5.1 - Fördelning av fiendetyper vid simulering av undvikande strategi.



Figur 5.2 - Fördelning av fiendeattributpoäng vid simulering av undvikande strategi.

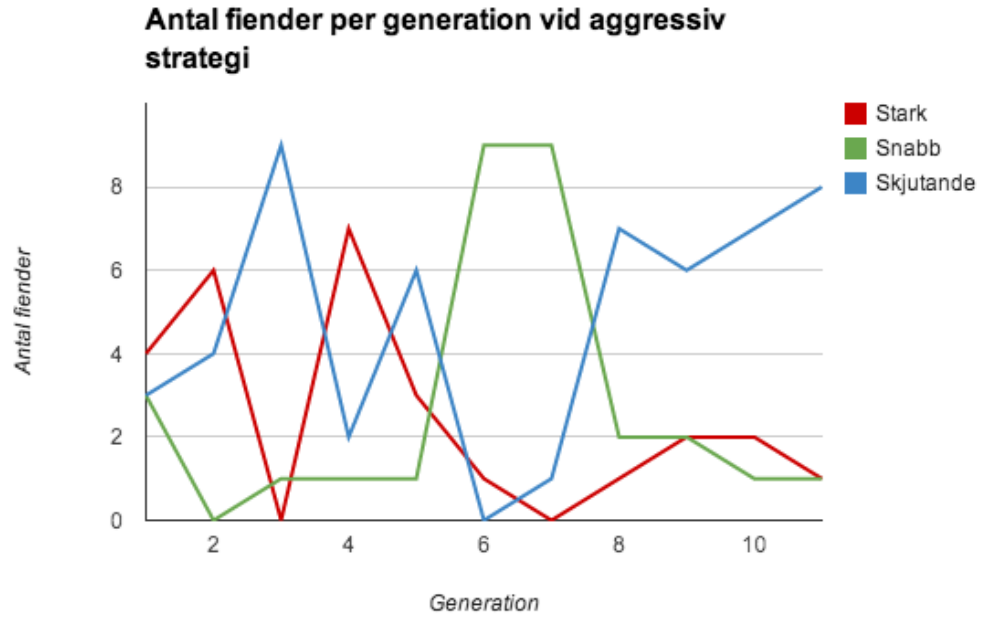
5.1.2 Aggressiv spelarstrategi

En aggressiv spelare använder sig av närstrid för att attackera fiender och satsar i denna utvärdering endast på styrka.

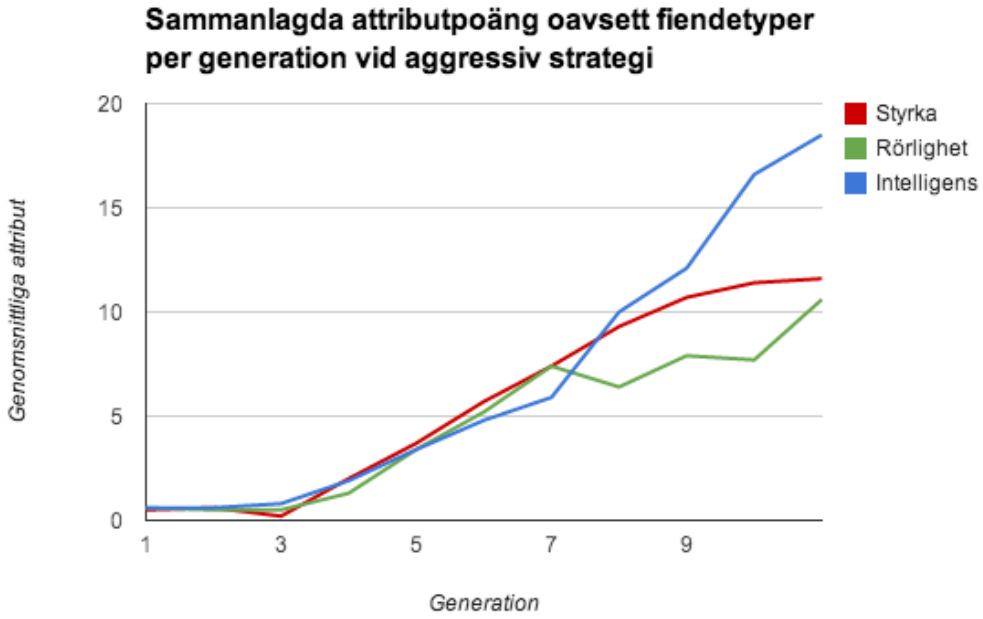
Fienderna förväntades inrikta sig på att skjuta och hålla sig på så långt avstånd från spelaren som möjligt. Dock visade det sig för denna strategi fördelaktigt för fienderna att inte bara utveckla sin förmåga att skjuta, utan även rörlighet då de även vill undvika närkamp med spelaren.

Figur 5.3 illustrerar fiendefördelningen utefter spelets nivåer. Då en aggressiv spelstil används innebär det att spelaren kommer ta skada från alla fiendetyper eftersom spelaren använder sig av sitt närstridsvapen. Som går att utläsa från grafen skiftar de olika fiendetyperna drastisk under spelets första 8 nivåer innan de avvaktande fienderna får övertag. Detta beror på att samtliga attribut var tydligt fördelaktiga till en början, då även starka fiender kunde göra mycket skada i närkamp, och fiendetyperna bestäms helt av de olika attributen. När spelaren börjar bli starkare avtar nyttan av styrka för fienderna, och andelen snabba och intelligenta fiender börjar öka.

De skjutande fiender som även har attributpoäng på rörlighet har en fördel, då de kan hålla sig på konstant säkert avstånd från spelaren och samtidigt avfyra projektiler. Detta kan tydligt utläsas från figur 5.4, som visar att trots en överväldigande andel skjutande fiender, kan en ökande förändring i styrka och rörlighet ses. Denna trend förutspåddes inte och ganska snart var de skjutande fienderna långt snabbare än spelaren vilket innebar att de lätt orsakade skada på spelaren samtidigt som de var svåra att döda.



Figur 5.3 - Fördelning av fiendetyper vid simulering av aggressiv strategi.



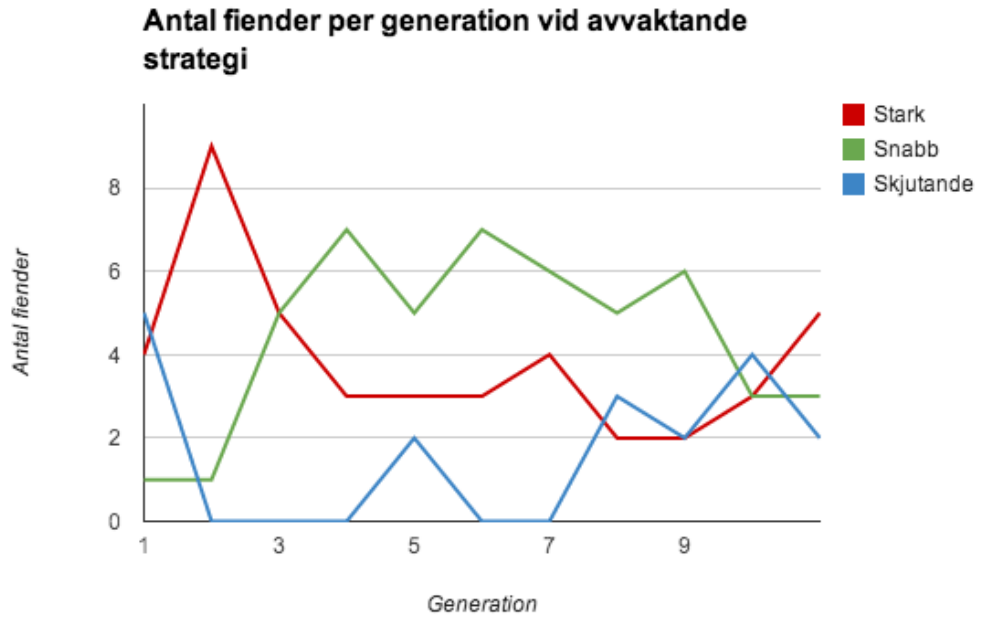
Figur 5.4 - Fördelning av fiendeattributpoäng vid simulering av aggressiv strategi.

5.1.3 Avvaktande spelarstrategi

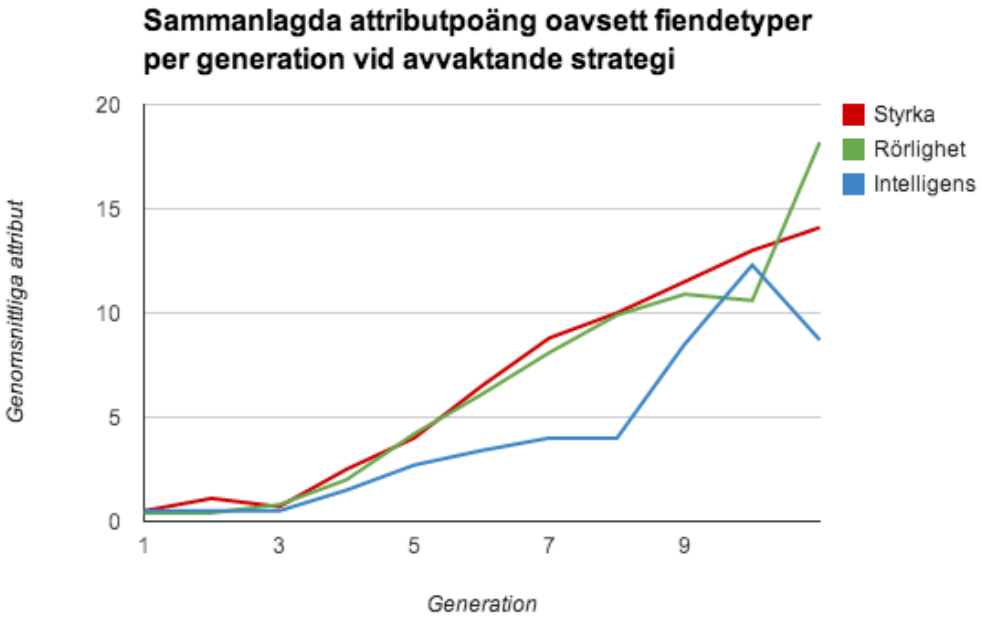
En avvaktande spelare satsar sina attributpoäng på intelligens och försöker använda sig av sitt långdistansvapen för att döda de attakerande fienderna.

Från figur 5.5 kan utläsas att det visat sig fördelaktigt för fienderna att fördela sina attribut på snabbhet, då den största utmaningen ligger i att komma ikapp den ständigt backande spelaren.

Till skillnad från en undvikande spelare så försöker en avvaktande spelare att skjuta ned sina fiender, vilket leder till ett ökat behov av styrka hos dessa. Alla fiendetyper har varit representerade i någon utsträckning, och först i slutet kan det i figur 5.6 ses hur intelligensen sjunker. Detta beror, precis som för en undvikande strategi, på att starka och rörliga fiender inte kan dra några fördelar av intelligens.



Figur 5.5 - Fördelning av fiendetyper vid simulering av avvaktande strategi.



Figur 5.6 - Fördelning av fiendeattribuspoäng vid simulering av avvaktande strategi.

5.2 Dynamisk användarupplevelse

Då de rigida strategimönstren som använts för statistikinsamling frångås kommer små avvikelser från strategin eller avsaknaden av en sådan skapa ett spel där motståndet dynamiskt förändras för att kontra spelaren. Det kommer för användaren märkas att spelet inte har ett förutbestämt mönster och på så sätt uppfylla målet med prototypen.

Använder sig spelaren av samma strategi över flera spelomgångar kommer ett mönster bland de olika fiendetyperna att uppstå. Spelaren kan då anpassa sig utifrån de mönster som går att se och byta strategi för att klara sig. Exempelvis kan en spelare spela på flera olika sätt under en och samma omgång och medvetet låta sig skadas av den fiendetyp denne har lättast att hantera, eller snabbt döda de som utgör störst hot. Resultatet blir då att de farliga fienderna inte får chansen att sprida sina gener till kommande generationer och spelaren möts till större usträckning av fiender som spelaren anser vara harmlösa.

5.3 Kodstatus

Protoypen består av två spellägen samt ett simuleringsläge vilka alla fungerar var för sig. I det första spelläget, som utvärderas i resultatet, jagar fienderna spelaren och försöker utplåna den.

Det andra spelläget består av en situation där spelaren istället jagar fienderna. Testning och utvärdering av det andra spelläget har nedprioriterats för att istället låta fokus ligga på första läget. Att se vilka konsekvenser som de genetiska algoritmerna ger och hur de påverkar spelupplevelsen i ett snarlikt scenario skulle ge ytterligare aspekter kring användningen av genetiska algoritmer. Det skulle troligtvis leda till fler intressanta resultat och ett bra underlag för att dra fler slutsatser för hur algoritmerna anpassar sig.

Det simuleringsläge som existerar i prototypen är tänkt att dels påvisa evolutionsbeteenden utifrån populationen, men även fungera som ett automatiskt testverktyg för att balansera och utvärdera produkten. För att tydligare visa resultatet kring simuleringen bör en mer omfattande beslutsfattning för fienderna implementeras, vilket inte har prioriterats. Som prototypen ser ut idag går det inte att utläsa relevant data ur simuleringsläget.

Kapitel 6

Diskussion

I följande avsnitt diskuteras ur ett tillbakablickande perspektiv implementeringen och resultatet av den spelprototyp som utvecklats under projektets gång. Kapitlet redovisar insikter och svårigheter som uppstått under arbetsgången samt reflektioner och tankar om ämnet i stort.

6.1 Konfigurering av genetiska algoritmer

Vid implementationen av prototypen testades flera olika målfunktioner för att på olika sätt bedömma hur väl en fiende klarat sig under en spelomgång. I slutändan valdes en mycket enkel utvärdering där fienderna endast bedöms efter hur mycket de lyckats skada spelaren. Det visade sig effektivare och lättare att balansera övriga parametrar till algoritmen samt själva prototypen över att försöka skriva en mer komplicerad målfunktion.

6.1.1 Enkel målfunktion

En enkel målfunktion kan leda till en lättexploaterad spelmodell; spelaren kan vilseleda den genetiska algoritmen genom att kontinuerligt byta strategi eller medvetet ta skada av fiender som bedöms ofarliga. Detta skulle kunna motverkats genom att skriva en mer komplex målfunktion.

Så som målfunktionen ser ut nu tar den endast hänsyn till hur mycket skada en fiende gjort på spelaren, en avancerad målfunktion skulle kunna ta hänsyn till fler faktorer än så. Ett exempel kan vara att en snabb eller stark fiende har befunnit sig inom ett kort avstånd från spelaren. Detta har inte bedömts som nödvändigt i prototypen då spelaren under utvärderingen hållt sig till en och samma strategi.

6.1.2 En generation

I prototypen kör den genetiska algoritmen en generation mellan varje spelomgång. Det gör att spelaren aldrig möts av samma population två gånger i rad. Ett problem som uppstod var att fiender utan fördelaktiga egenskaper kunde få bra utvärderingsresultat av tillfälligheter så som enkla misstag av spelaren.

En lösning som undersöktes för att undvika att enkla misstag gav stora utslag var att låta varje generation leva vidare under flera omgångar, för att sedan evaluera dem efter sin genomsnittliga prestation. Detta visade sig dock utveckla fienderna i en takt som var allt för långsam. Givet en stabilare och mer välplanerad utgångspunkt för individerna skulle möjligen en lägre utvecklingstakt kunna hanteras bättre.

6.1.3 Mutationstakt

Ett återkommande problem vid balanseringen av prototypen var att motståndet ibland stagnerade då algoritmerna misslyckas med att hitta utmanande fiender. En effektiv lösning på detta visade sig vara att radikalt öka mutationstakten efter en runda där motståndet bedömdes ha varit enkelt. På detta sätt ökas chansen att fördelaktiga egenskaper introduceras i den nya populationen. Resultatet av åtgärden var att två generationer i rad sällan erbjöd ett svagt motstånd och övriga balansåtgärder gjorde att det ofta enbart krävdes en eller två generationer för att motståndet ska börja öka mer stabilt igen.

6.2 Insikter

Under arbetets gång har aspekter och möjligheter med genetiska algoritmer i spel som inte var tydliga vid projektets början visat sig.

6.2.1 Dynamik utan avancerad artificiell intelligens

Artificiell intelligens är ett centralt område inom spelutveckling som får mycket utrymme tack vare den positiva effekt en ökad dynamik kan ge spelupplevelsen. Genetiska algoritmer skulle kunna möjliggöra en liknande effekt av adderad dynamik utan den komplexitet artificiell intelligens ofta ger upphov till. Exempelvis skulle avancerade koncept som självlärande beteenden hos fiendeenheter troligtvis kunna ersättas av genetiska algoritmer.

6.2.2 Oväntade resultat vid spelarstrategi

Under prototypanalysen framkom att det vid en aggressiv närstridsstrategi från spelaren inte var fördelaktigt för fienderna att bara ha attributpoäng distruberade på intelligens, utan också rörlighet.

Fenomenet som uppstod var att de projektilavfyrande fienderna efter några generationer började distribuera attributpoäng på rörlighet för att kunna hålla sig på konstant avstånd från spelaren; med en välavvägd balans mellan de båda kunde fiendetypen effektivt attackera spelaren utan att själv ta skada.

Detta visar på den typ av oväntade positiva fenomen som en effektiv användning av genetiska algoritmer kan ha. Enheterna utvecklades först som väntat emot det attribut dess förmågor var utvecklade för, men skiftade sedan delvis distributionsbalans. Detta resulterade i enheter som var effektivare än vad som i förhand hade kunnat förutspås.

6.2.3 Inflation av attribut

Från resultatet av hur fienderna reagerar på olika spelarbeteenden kan inflation påvisas. Inflation i detta sammanhang innebär att fienderna får liknande attribut som spelaren, vilket i sin tur medför att värdet i att vara stark i detta attribut minskar. Ett konkret exempel på detta är när spelaren fördelar sina attributpoäng för att bli så snabb som möjligt och fienderna

då följer samma utveckling. På detta sätt försvinner mycket av den fördel poäng i attributet var tänkt åstadkomma.

6.2.4 Evolverande val av beteende

Prototypen som utvecklats i detta arbete hanterar enheter med statiska kombinationer av förmågor och beteenden; en projektilavfyrande enhet har alltid samma beteende utformat efter dess egenskaper. Dock finns även möjligheten att inkludera beteendeattribut vid evaluering. Exempelvis skulle flera olika bestämda beteenden kunna utvecklas parallellt med ett antal förmågor och dessa utvecklingar skulle kunna ske under programmets exekvering. I fallet med vår prototyp skulle då rimligtvis beteendena utformade för en viss fiendetyp snabbt kombineras med detta.

I ett mer komplicerat exempel kan man tänka sig att olika beteenden kombineras med olika egenskaper beroende på en användares spelstil. Detta skulle med bara ett fåtal egenskaper och beteenden resultera i en relativt stor mängd möjliga kombinationer. Ett spelarbeteende skulle då i teorin kunna bemötas med flera olika typer av respons utan att varje typ implementeras statiskt för sig. Osäkerheten och variationen i resultaten för denna metod gör den dock mer lämpad för att använda i test- och simuleringssyfte. Vi gör ändå bedömningen att det i välkontrollerade enklare fall skulle fungera bra att använda denna teknik i en slutprodukt.

6.3 Svårigheter och utmaningar

Många av de utmaningar som förväntades hanterades relativt enkelt samtidigt som flera oförutspådda svårigheter tog mycket fokus under arbetet.

6.3.1 Balanseringssvårigheter mellan fiendeenheter

Ett av de största problemen som försvårade utvecklingen genom hela projektet var att fiendeenheterna med sina respektive förmågor var mycket svåra att balansera ur ett svårighetsgradsperspektiv. Att utveckla

beteende- och egenskapsmässigt olika men samtidigt likvärdigt utmanande enheter är en stor utmaning. De enheter vars beteende och förmågor relativt övriga var tydligt starkare fick ofta ett direkt övertag.
Då tidsbristen förhindrade att mer fokus lades på denna typ av balansering blev det slutgiltiga resultat inte fullt så avvägt som önskat och en viss obalans mellan fiendeenheterna är påtaglig. Så behöver dock inte fallet vara i en mer omfattande spelimplementation då denna aspekt ofta är högt prioriterad och i regel fungerar mycket bra. Ett effektivare tillvägagångssätt för detta projekt skulle kunna ha varit att applicera genetiska algoritmer på en färdig välbalanserad spelmodell.

6.3.2 Stor variation av fiendetyper mellan omgångar

I resultatet kan utläsas hur den totala distribueringen av fiendetyper vid olika spelstrategier hos spelaren oftast väger över åt något håll. Över många spelade omgångar så framställs viktningen vara tydlig i någon riktning, men faktum är att mellan två separata spelomgångar kan fördelnigen mellan fiendetyper skilja sig drastiskt. Om en fiendetyp lyckats bra under en omgång kan den sprida sig explosionartat över populationen till kommande omgång.

Detta beror på att varje fiende utvärderas efter varje omgång, och att korsning applicerats. Då korsning snabbt kan sprida egenskaper över hela populationen, speciellt då populationen endast består av 10 individer, blir resultatet ibland en drastisk förändring i balansen mellan olika fiendetyper. Då den genetiska algoritmen har stort spelrum i prototypen ses ofta stor förändring på kort tid.

För att begränsa denna häftiga balansskiftning skulle en begränsning av den genetiska algoritmen vara nödvändig vilket i sin tur på ett sätt skulle motverka syftet med implementeringen. Finjustering av de olika parametrarna till algoritmen skulle delvis kunna lösa detta problem.

6.3.3 Genetiska algoritmer och testning

I sin oförutsägbarhet är genetiska algoritmer svåra att testa på ett effektivt sätt. Under detta arbete gjordes all testning relaterad till den genetiska algoritmen för enkelhetens skull genom faktiska körningar av spelet. Metodiken fungerade acceptabelt i detta fall men i ett verkligt scenario kan detta tillvägagångssätt avfärdas som osäkert och tidmässigt ineffektivt.

Då den genetiska algoritmen hela tiden utvärderas från en målfunktion kan det ses som att algoritmen i viss mån automatiskt genomför testning under körning. Det innebär att fokus av testningen flyttas från algoritmen till målfunktionen. Målfunktionen i sig kan vara svårdefinierad, speciellt i mer omfattande applikationer, vilket i sin tur gör den svårtestad.

Även om algoritmerna inte är omöjliga att testa talar denna aspekt emot effektiv användning i mer omfattande implementationer.

6.4 Framtida kommersiell användning

Sett från våra resultat finns inga prestandamässiga problem med att använda genetiska algoritmer i spelutveckling och köra dessa vid exekvering. Dock är det svårt att förutspå och kontrollera resultaten dessa ger.

Lösningen skulle kunna vara att inte ge algoritmerna för stort utrymme och begränsa tillåtna resultat. Istället för att som i detta arbete låta algoritmen generera attribut för fiendeenheter skulle man kunna tänka sig att den endast väljer vilken fiendetyp som ska genereras utifrån ett antal färdiga val. Detta reducerar potentialen för oväntade resultat men behåller mycket av den lättimplementerade dynamik som algoritmerna möjliggör.

Kapitel 7

Slutsats

Den genetiska algoritmen som implementerats i prototypen har fungerat väl och uppfyllt sitt syfte. Olika spelarstrategier har tydligt genererat skiftande respons från spelet. I begränsad form bedöms genetiska algoritmer kunna användas i realtid även för kommersiellt bruk inom spelutveckling. Genom att kontrollera och begränsa algoritmernas spelrum kan problem som uppkommer i och med deras oförutsägbarhet undvikas.

Källor

Biles, J.A.B. (1994) *GenJam: A Genetic Algorithm for Generating Jazz Solos*. Rochester Institute of Technology. http://igm.rit.edu/~jabics/GenJam94/Paper.html. (2014-04-27).

Charles, D.C., Fyfe, C.F., Livingstone, D.L., McGlinchey, S.M., *Biologically Inspired Artificial Intelligence for Computer Games*. Hershey, United States of America: IGI Publishing, 2008.

Davey, R.D (2014), *Readme: Introduction*. Phaser GitHub Repository. https://github.com/photonstorm/phaser. (2014-05-06).

Denzinger, J.D., Loose, K.L., Gates, D.G and Buchanan, J.B. (2005), Dealing with parameterized actions in behavior testing of commercial computer games. I *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG)* , 4-6 april, 2005, Colchester . s. 37-43.

Haupt, R.L.H., Haupt, S.E.H. (2004), *Practical Genetic Algorithms*, Second edition. Hoboken, New Jersey: John Wiley & Sons, Inc.

Lucas, S.M.L., Kendall, G.K. (2006), *Evolutionary Computation and Games*. IEEE Computational Intelligence Magazine. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1597057. (2014-05-08).

del Notario, C.B.P.N., Baert, R.B., D'Hondt, M.H. (2012), Multi-Objective Genetic Algorithm for Task Assignment on Heterogeneous Nodes. *International Journal of Digital Multimedia Broadcasting*. http://www.hindawi.com/journals/ijdmb/2012/716780/. (2014-05-06)

Schwaber, K.S., Sutherland, J.S. (2013), *The Scrum Guide*. https://www.scrum.org/Scrum-Guide. (2014-05-12).

Wahde, M.W. (2008), *Biologically inspired optimization methods, an introduction*. Southampton, Boston: WIT Press

Appendix A

Spelidé

Spelaren får ta rollen som Charles Darwin, den legendariske biologen, strandsatt på en ö i Galapagos. Ständigt under attack från olika evolverande djurformer får han utveckla och anpassa sina färdigheter för att överleva sin tid på ön.

Grundidén är att utveckla ett 2d-spel där spelaren ses ovanifrån. Spelaren kommer spela igenom ett visst antal nivåer, där varje nivå består av en ny våg av genererade fiender. Spelaren klarar en nivå genom att antingen döda alla fiender eller överleva en viss tid.

Nivåerna kommer att bestå utav enklare landskap med varierande mängd vegetation som spelaren inte kan korsa. I en skogsmiljö skulle detta kunna motsvaras av objekt som träd, buskar och stenar. Nivåerna kommer även att ha någon form av gräns som stänger in spelaren och fienderna i ett begränsat område. Då Darwin befinner sig på en ö kommer troligtvis denna gräns i de flesta fall att utgöras utav en strand.

Fienderna kommer ha flera olika sätt att attackera spelaren, exempelvis skjuta på avstånd eller slåss nära beroende på vapen. Mellan nivåerna kommer fienderna utvärderas för att sedan anpassas utifrån vad som fungerade bäst. Detta kan avgöras av hur länge en fiende överlever eller hur mycket skada den hinner åsamka spelaren innan den blir besegrad. I de inledande nivåerna kommer fiendernas egenskaper att slumpas fram.

Mellan varje nivå kommer spelaren kunna använda sina resurser/poäng för att förbättra ett attribut, exempelvis styrka, snabbhet eller intelligens. Spelaren kommer få välja själv när nästa våg skall startas och har därför tid på sig att köpa attribut eller kanske spara resurser till nästa nivå. Beroende på hur spelaren fördelar sina poäng, kommer nästa fiendevåg att anpassas efter dessa.

Finns tid kommer olika kartor implementeras, så att spelaren kan komma till en ny karta efter ett visst antal nivåer. Kartorna kan även användas till att balansera spelet. Ifall spelaren spenderat sina attribut på att till exempel överleva istället för att döda fiender kommer vissa kartor vara utformade med återvändsgränder för att göra det svårare.

Appendix B

Kod för den genetiska algoritmen

```
'use strict';

Darwinator.GeneticAlgorithm = {

  // GA parameter constants
  POPULATION_SIZE:          10,
  CROSSOVER_PROBABILITY:    0.8,
  MUTATION_PROBABILITY:     0.025,
  TOURNAMENT_PARAMETER:     0.75,
  NUMBER_OF_GENERATIONS:    100,
  NUMBER_OF_GENES:          3,
  TOURNAMENT_SIZE:          4,
  ELITISM_DEGREE:           2,

  // GA parameter variables
  variableRange:           undefined,
  mutationRate:            undefined,

  // Other constants
  PLAYER_ADVANTAGE:         10,
  POOR_MAX_FITNESS:         0.1,

 /**
  * Generates a population of individuals from a given population.
  * The new population is likely to be more adapted to find a solution
  * to the goal function.
  * @method Darwinator.GeneticAlgorithm#generatePopulation
  * @param {Array}  - Array containing the chromosomes from which to start.
  *                     Optionally can contain the fitness-levels
  *                           if the chromosomes were pre-evaluated.
  *
  * @param {Object} - The options for the algorithm. Options are:
  *                   preEvaluated - True if the population is already evaluated.
  *                                       A pre-evaluated population will
  *                                       only run for one generation
  *                  varRange     - The sum which the chromosomes attributes should equal.
  * @return {Array} The new wave of enemies as chromosomes
  */
  generatePopulation: function(population, options, targetFunction) {
    // If no population is given, randomize a new one and return it
    if(!population.length){
      population = this.initPopulation(options.varRange);
      return population;
    }

    // weak population => more mutation
    this.mutationRate = population.maxFit <= this.POOR_MAX_FITNESS ? 20 : 1;

    // algorithm main loop
```

```
  for (var i = 0; i < (options.preEvaluated ? 1 : this.NUMBER_OF_GENERATIONS); i++) {

    if (!options.preEvaluated) {
      evaluatePopulation(population, targetFunction);
      population.bestInd = population[0];
      population.maxFit = population.bestInd.fitness;
    }

    var tmpPopulation = [];
    tmpPopulation.fitness = population.fitness.slice(0);

    // selection
    for(var l = 0; l < population.length; l += 2) {
      var index1 = this.selection(population.fitness);
      var index2 = this.selection(population.fitness);
      // crossover
      if (Math.random() < this.CROSSOVER_PROBABILITY) {
        var chromePair        = this.cross(population[index1], population[index2]);
        tmpPopulation[l]      = chromePair[0];
        tmpPopulation[l + 1]  = chromePair[1];
      } else {
        tmpPopulation[l]      = population[index1];
        tmpPopulation[l + 1]  = population[index2];
      }
      // mutation
      tmpPopulation[l]    = this.mutate(tmpPopulation[l]);
      tmpPopulation[l+1]  = this.mutate(tmpPopulation[l+1]);
    }

      // elitism
    for(l = 0; l < this.ELITISM_DEGREE; l++) {
      tmpPopulation[l] = population.bestInd;
    }

    // replace old population
    population = tmpPopulation;
  }

  return population;
},

/**
* Generates a generation of chromosomes with randomized genes scaled
* by the specified number.
*
* @method Darwinator.GeneticAlgorithm#initPopulation
* @param {Number} - The sum which the chromosomes attributes should equal.
* @return {Array} A population of chromosomes with randomized values.
*/
initPopulation: function(variableRange) {
  var population, i, l;
  population = [];
  for (i = 0; i < this.POPULATION_SIZE; i++) {
    population[i] = [];
    for (l = 0; l < this.NUMBER_OF_GENES; l++) {
      population[i][l] = Math.random() * variableRange;
    }
```

```
  }

  return population;
},

/**
* Cross two individuals to create two new ones by performing a tradeoff between
* two randomly selected genes in each individual.
*
* Example: Select index of x and y.
* First individual: decrease x and increase y
* Second individual: decrease y and increase x
*
* @method Darwinator.GeneticAlgorithm#cross
* @param {Array} [firstInd] - The first individual to be crossed
* @param {Array} [secondInd] - The second individual to be crossed
* @return {Array} - A tuple containing the two new individuals.
*/
cross: function(firstInd, secondInd) {
  var crossPoint1 = Math.round(Math.random()*(this.NUMBER_OF_GENES - 1));
  var crossPoint2 = Math.round(Math.random()*(this.NUMBER_OF_GENES - 1));

  var tradeOff        = Math.random();
  var tradeOffAmount1 = Math.round(tradeOff * firstInd[crossPoint1]);
  var tradeOffAmount2 = Math.round(tradeOff * secondInd[crossPoint2]);

  var newInd1 = firstInd.slice(0);
  var newInd2 = secondInd.slice(0);

  newInd1[crossPoint1] -= tradeOffAmount1;
  newInd1[crossPoint2] += tradeOffAmount1;

  newInd2[crossPoint2] -= tradeOffAmount2;
  newInd2[crossPoint1] += tradeOffAmount2;

  return [newInd1, newInd2];
},

/**
* Select a individual based on the fitness levels of the entire population.
* this.TOURNAMENT_SIZE many individuals will be selected at random,
* and one of them will be returned. The tournament works in such a way
* that a individual with a high fitness level is more likely to be returned
* from the tournament, but is not more likely to participate.
*
* @method Darwinator.GeneticAlgorithm#selection
* @param {Array} - The fitness levels of the entire population
* @return {Number} - The index of the selected individual.
*/
selection: function(fitnessLevels) {
  var tournamentParticipants = [];

  /* Select individuals at random, and represent them as a tuple
  * containing their indexes and their fitness levels. */
  for (var i = 0; i < this.TOURNAMENT_SIZE; i++) {
    tournamentParticipants[i]    = [];
    tournamentParticipants[i][0] =
```

```
              Math.round(Math.random() * (this.POPULATION_SIZE - 1));
      tournamentParticipants[i][1] = fitnessLevels[tournamentParticipants[i][0]];
    }

    /* Sort by fitness levels */
    tournamentParticipants.sort(
      function(a,b) {return b[1]-a[1];}
    );

    for(i = 0; i < this.TOURNAMENT_SIZE; i++) {
      if (Math.random() < this.TOURNAMENT_PARAMETER) {
        return tournamentParticipants[i][0];
      }
    }

    return tournamentParticipants[i - 1][0];
  },

  /**
  * Mutate a given individual by performing a tradeoff between
  * two randomly selected genes.
  *
  * @method Darwinator.GeneticAlgorithm#mutate
  * @param {Array} - The individual to be mutated
  * @return {Array} - The mutated individual.
  */
  mutate: function(individual) {
    if (Math.random() < this.MUTATION_PROBABILITY * this.mutationRate) {
      var gene1 = Math.round(Math.random()*(this.NUMBER_OF_GENES - 1));
      var gene2 = Math.round(Math.random()*(this.NUMBER_OF_GENES - 1));
      var tradeOffAmount = Math.round(Math.random() * individual[gene1]);

      individual[gene1] -= tradeOffAmount;
      individual[gene2] += tradeOffAmount;
    }

    return individual;
  },

  /**
  * Evaluates an individual based on the given target function.
  * A higher target function value will give a higher fitness.
  *
  * @method Darwinator.GeneticAlgorithm#evaluateInd
  * @param {Array} - The individual to be evaluated
  * @return {Number} - The fitness level of the individual
  */
  evaluateInd: function(individual, targetFunction) {
    var fitness = 1 - (1 / targetFunction(individual));
    return fitness === -Infinity ? 0 : fitness;
  }

};
```

Appendix C

Kod för konvertering mellan fiende och kromosom

```
'use strict';

Darwinator.Helpers = {

/**
  * Translates an enemy group of sprites to chromosomes and fitness levels.
  *
  * @method Darwinator.Helpers#enemiesToChromosomes
  * @param {Phaser.Group} - A group of enemy sprites
  * @param {Number} - The number the sum of an individual's attributes should equal.
  * @return {Array} - An array of chromosomes. Also contains an array of fitness-levels.
  *
  */
  enemiesToChromosomes: function(enemyGroup, varRange) {
    // Function for converting a single Phaser.Sprite into a chromosome
    var enemyToChromosome = function (enemy, varRange) {
      var chrom, attrSum, scale, i;
      attrSum = enemy.attributes.strength + enemy.attributes.agility
                                          + enemy.attributes.intellect;
      chrom = [enemy.attributes.strength, enemy.attributes.agility,
                                          enemy.attributes.intellect];

      // Factor to scale with to reach the variable range
      scale = varRange / attrSum;

      for (i = 0; i < chrom.length; i++) {
        chrom[i] *= scale;
      }

      return chrom;
    };

    var currentSize, population, i, enemy;
    currentSize = enemyGroup.length;
    population  = [];
    population.fitness = [];
    population.maxFit = -Infinity;
    for(i = 0; i < currentSize; i++){
      enemy = enemyGroup.getAt(i);
      population[i] = enemyToChromosome(enemy, varRange);
      // Evaluate with external goal function
      population.fitness[i] = Darwinator.EVALUATE_ENEMY(enemy);
      if (population.fitness[i] > population.maxFit) {
        population.bestInd = population[i];
        population.maxFit = population.fitness[i];
      }
    }
    return population;
  },
```

```
  /**
  *  Given the values of their attributes, spawns enemy sprites
  *  and adds them to the game.
  *
  * @method Darwinator.Helpers#chromosomesToSprites
  * @param {Array} [population] - The chromosomes of the generation of themies.
  * @param {Phaser.Group} [enemyGroup] - The enemy sprite group in which
  *                                      to place the enemies.
  * @param {Array} [spawnPositions] - The positions on which the enemies
  *                                      are allowed to spawn.
  * @return {Phaser.Group} The new group of enemies.
  */
  chromosomesToSprites: function(population, group ,spawnPositions){
    var i, pos, strength, agility, intellect, enemy;
    for(i = 0; i < population.length; i++) {
      pos         = spawnPositions[i % spawnPositions.length];
      strength    = population[i][0];
      agility     = population[i][1];
      intellect   = population[i][2];
      enemy       = new Darwinator.Enemy(group.game, undefined, pos.x, pos.y,
                                      undefined, strength, agility, intellect);
      group.add(enemy);
    }

    return group;
  }

};
```

Appendix D

Insamlad data om fiendetyper och attributfördelning

Undvikande strategi:				
Generationsutveckling:		Stark	Snabb	Skjutande
	1	5	4	1
	2	4	5	1
	3	9	0	1
	4	10	0	0
	5	9	1	0
	6	9	1	0
	7	5	5	0
	8	4	6	0
	9	1	8	1
	10	2	8	0
	11	2	7	1
Attributsfördelning:		Styrka	Rörlighet	Intelligens
	1	0.6	0.6	0.5
	2	0.8	0.7	0.3
	3	1.1	0.6	0.2
	4	3	1.5	1.4
	5	4.8	3.9	2.2
	6	6.8	5.9	3.2
	7	7.6	8.8	4.4
	8	8.9	12	5
	9	9.2	16.1	5.6
	10	11.8	17.5	5.6
	11	10.8	21.9	8.2

Aggressiv Strategi				
Generationsutveckling		Stark	Snabb	Skjutande
	1	4	3	3
	2	6	0	4
	3	0	1	9
	4	7	1	2
	5	3	1	6
	6	1	9	0
	7	0	9	1
	8	1	2	7
	9	2	2	6
	10	2	1	7
	11	1	1	8
Attributsfördelning:				
		Styrka	Rörlighet	Intelligens
	1	0.5	0.6	0.6
	2	0.6	0.5	0.6
	3	0.2	0.5	0.8
	4	2	1.3	1.9
	5	3.7	3.4	3.4
	6	5.7	5.2	4.8
	7	7.4	7.4	5.9
	8	9.3	6.4	10
	9	10.7	7.9	12.1
	10	11.4	7.7	16.6
	11	11.6	10.6	18.5

Avvaktande strategi:				
Generationsutveckling:		Stark	Snabb	Skjutande
	1	4	1	5
	2	9	1	0
	3	5	5	0
	4	3	7	0
	5	3	5	2
	6	3	7	0
	7	4	6	0
	8	2	5	3
	9	2	6	2
	10	3	3	4
	11	5	3	2
Attributsfördelning:		Styrka	Rörlighet	Intelligens
	1	0.5	0.4	0.5
	2	1.1	0.4	0.5
	3	0.7	0.8	0.5
	4	2.5	2	1.5
	5	4	4.2	2.7
	6	6.5	6.1	3.4
	7	8.8	8.1	4
	8	10	9.9	4
	9	11.5	10.9	8.5
	10	13	10.6	12.3
	11	14.1	18.2	8.7