Evolving Intelligent Agents

by

Blain Maguire, B.Sc.(Hons)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2012

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Blain Maguire

August 29, 2012

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Blain Maguire

August 29, 2012

Acknowledgments

I would like to thank my supervisor Saturnino Luz for his guidance throughout the project. I would also like to thank my close friends and family for their support. Special thanks to the course director John Dingliana who in addition to providing feedback has been remarkably helpful throughout the year.

Blain Maguire

University of Dublin, Trinity College
August 2012

Evolving Intelligent Agents

Blain Maguire
University of Dublin, Trinity College, 2012

Supervisor: Saturnino Luz

There is no question that over the years computer games have become increasingly more realistic looking due to numerous improvements in both hardware and software. Commercial games from large studios especially continue to push the envelope in this regard. Despite the increase in capability though, artificial intelligence in games has yet to see such a considerably noticeable increase.

One of the areas which has some room for innovation is the actual process of actually creating intelligent agents in games. While various libraries and frameworks exist to make life a easier for developers, all too often, core agent behavior is still being hand coded. Couple that with the fact that artificial players are often seen as an low on the list of priorities in actually shipping a commercial game and it leaves much to be desired.

It is the aim of this research project to explore the use of evolutionary computation, itself a form of machine learning, in order to create agents which to a human observer, appear intelligent. A series of computer games to use as a testbed have been developed

in order to evaluate the feasibility of this. Each of the games have a headless (no graphics) mode which allow for less computational overhead when evaluating individual agents. A framework for evolutionary computation was integrated and used for this. Custom programming languages have also been developed for the agents, which allow for game specific instructions relating to sensing and interacting with the various games. Given sufficient time to evolve, agents completed the levels of all the various games. Watching agents play, their behaviour appears intelligent. Encouraging results suggest further potential research, from improving game bot behaviour to automatic testing of levels in games.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past few decades computer games have become increasingly complex and realistic looking over time, so much so that we have become accustomed to expecting such increases with the release of new games. Graphics processing units have become commonplace, their capabilities have grown tremendously over the years. Software utilizing them has matured considerably, making it easier for developers to get the most out of these devices. There also have been some notable advances such as the use of procedural generation of terrain [1] and as well as man-made structures, which assist in the actual creation of such realistic graphics, both in quality and quantity.

From the start of the latter half of the last decade, there has also been drastic change in how we interact with computer games. The controller, the keyboard and mouse may all be around for a long time yet, but certainly the possibilities for different, more intuitive and immersive experiences have opened up. Starting with the Wii, continuing on with the Kinect and the Move, these devices have brought with them a drastic change in how we perceive how we can interact with games. Even on mobile, multi-touch technology and various sensors such as accelerometers is making game developers reconsider interaction with their games.

With these advancements in mind, given that a lot of games feature interactive agents of some kind, it opens up new possibilities. It is arguable that for experiences involving these agents to be highly entertaining that they need not exhibit a high degree of artificial intelligence or autonomy. While this may in fact be quite true, it still doesn't rule out the possibility for great improvements in this area. Just like improved

graphics or interfaces may not necessarily be in and of themselves be necessary for an entertaining game, they have no doubt provided value and nevertheless raised the expectations of gamers over time.

It remains to be seen where exactly the area of agent artificial intelligence in games could be drastically improved in the future, both in their quality and how they are developed. There is no question that there certainly have been improvements over the years. Open source and commercial solutions do exist already for common problems faced by developers such as navigating multiple non-player characters around levels simultaneously. This project does not seek out to solve or improve on any of these well known problems regarding agents. Instead, the focus is more on how taking a very different approach regarding the development of these agents can yield interesting and promising results.

1.1 Motivation

In order to develop agent behavior for a specific game, development could entail a mix of using various libraries or middleware which can be brought to bear on the more common game specific problems. This is good for a number of reasons, the most obvious being a saving on development time and costs. Certain assurances also come with tried and tested code. For more specific things relating to agent behavior, such as game specific strategies or personalities, it's not unusual for such things to be coded by developers themselves. High level programming languages are usually preferable in these cases. It allows developers to spend more time focusing on the problems at hand at a high level and avoid getting caught up worrying too much about the lower level details. Optimizing for performance may still be a concern but it's likely to be less of a priority when compared with something which is more computationally expensive.

When we look at another area, like computer graphics for example, great tools and software exist for enhancing the look and visual appearance of games. There is also a good existing body of software available to create procedurally generated content and it is being used to great effect in games. While procedurally generated content may not trump that created by professional artists any time soon in every regard, it does still provide utility in several ways. Procedurally generated textures can be used on natural objects. Vast three dimensional scenic landscapes and urban environments can

also be generated procedurally and used as a base for artists to build on and improve on. Ultimately, such procedural techniques save primarily on time and effort. Even when time has been invested to create custom tools to do the job, creating additional unique content can simply be a matter of tweaking parameters. Combine this with the sheer scale of content which can be created in this manner and you have something which is very powerful indeed.

With this in mind, in a similar vein, there is also the prospect of procedurally generating agent logic and behavior, with a mind to similar benefits. This project will seek to investigate how evolutionary computation in particular can be used to do just that. Aside from the benefit of saving on development time, this project also seeks to explore and highlight other potential benefits surrounding this technique which may arise from using it. An ideal testbed for demonstrating what is capable, would be to actually use some games and evolve various agents that can play them. This was one of the primary aims of this project from the very beginning.

1.2 Outline

This document will start by providing some background to the technique which will be used to evolve agents as well as providing a detailed review of some of the techniques which are currently considered state of the art in this regard. The overall approach, including key design decisions which were made for the project are explored as well as the rationale behind them. This is then followed with a detailed review of the important implementation details. Following on from this, an in-depth evaluation is given on how the project itself was assessed. Finally some conclusions will be drawn from what has been done, in light of results. Criticism and also possibility for future work will also be discussed.

Chapter 2

Background and State of the Art

2.1 Introduction

Genetic programming (GP) is a form of evolutionary computation which has been used extensively used throughout this project. The general idea is to evolve computer code. Code can be represented in different ways. It can also be mutated or recombined. Good solutions are selected based on one or more criteria which can be evaluated (by means of a fitness function). Over time, better solutions emerge.

Its origins can traced back to the early 1950s. It began with Nils Aall Barricelli[2] applying evolutionary algorithms to evolutionary simulations. There have been many developments since then. This includes more formalization, new applications as well as modifications and extensions. However, it's worth pointing out that the fundamental principles of this evolutionary algorithm-based methodology and its biologically inspired roots remain the same.

Various algorithms that might fall under the general category Genetic Programming were described in the 1980s. The closest of these to what we might consider a modern GP would be that of Cramer [3]. However it was the research of Koza [4] which defined the field more clearly and established it as an important sub-field of evolutionary computation. The type of GP which has arisen as a result has since been the archetype and is still widely used to this day.

Grammars are a core structure for representation in the field of Computer Science and they are widely used. It is no surprise then to find their use in GP. Initially this

grammar based approach formed only a small part of GP. This has since been expanded on to where Grammatical Evolution is now one method which is widely applied. This method brings us to Grammar Guided Genetic Programming (GGGP). A detailed survey of this area alone is given by [5]. Research in the area is still on-going, for example a recent new approach for generation of constants for evolutionary grammars was taken by [6] in 2011.

Later in the 1990s the first fully-fledged systems would use this and this would lead on to areas of research which are more current. It's also worth pointing out that research and development into this area continues to this day. For example in 2007, Garcia et al [7] have shown an algorithm for generating grammars which does affect the rate of convergence of GP on a solution which they used a real-world task of breast cancer prognosis.

2.2 Modern Applications of Genetic Programming

As mentioned in the previous section, GP has found its way into cancer research. [8] details some of the applications in that field.

In Koza's Genetic Programming IV: Routine human-competitive machine intelligence [9] he presents the application of GP to a wide variety of problems involving automated synthesis of controllers, circuits, antennas, genetic networks, and metabolic pathways.

The book describes fifteen instances where GP has created an entity that either infringes or duplicates the functionality of a previously patented 20th-century invention, six instances where it has done the same with respect to post-2000 patented inventions, two instances where GP has created a patentable new invention, and thirteen other human-competitive results.

A contest which is held yearly since 2004 at the Genetic and Evolutionary Computation Conference called the annual "humies" awards for human competitive results. This is in the same spirit as Koza's book. Some notable entries for 2011 include evolving a computer player for the computer card game FreeCell [10] which won that year.

2.3 Modern Theory of Genetic Programming

In Foundations of Genetic Programming [11] more mathematical and empirical analyses of GP are given. The book Genetic Programming Theory and Practice [12], although not strictly about theory, contains a lot of content from leading theorists on the subject and ties it in well with topics in other fields such as biology (population genetics for example) which may be of interest.

Holland's Schema Theorem [13] is known as a more formal way of describing why genetic algorithms work. Put simply, it works by describing how a sequence of binary strings come to converge on a solution. Schema theory also exists for GP for variable length strings [14].

Goldberg, Deb and Clark[15] have also talked about how noise/population variance and population size affects the rate of convergence. In the paper they add noise to the various functions such as fitness or selection and detail the results as well as parameters such as the population size. A key stated conclusion is that at low population sizes the genetic algorithm makes many errors of decision, and the quality of convergence is largely left to the vagaries of chance.

2.4 Alternatives and Comparisons within Machine Learning

A wide variety of alternatives to GP which could be applied on the same problem, each with their own advantages and disadvantages. It's also worth pointing out that it isn't the only means of creating programs by means of evolutionary computation. For example, [16]presented a new technique for constructing programs through Ant Colony Optimization.

A recent paper [17] which did a a comparative analysis of GP and Artificial Neural Networks for metamodeling of discrete-event simulation (DES) models. The results of the study showed that GP provides greater accuracy in validation tests. However it is worth noting that GP was more computationally intensive during the metamodel development stage.

2.5 Parallelism and Genetic Programming on Modern Hardware

There has been a lot of development in recent years getting GP to work on graphics cards. The fitness function is often the most costly in genetic programming and genetic algorithms. Research such as [18] which is in the direction of making is it less expensive computationally is of great benefit. The cost incurred is on individuals and any savings in that regard is multiplied by way of having to evaluate a population. The ability to evaluate individuals in a given population in parallel also offers a tremendous performance boost in that the time to reach the next generation is reduced as multiple individuals are being evaluated in parallel. Langdon [19] and Harding [20] explore this idea with practical implementations in detail.

Cartesian Genetic Programming [21], which first appeared in the early 1990s, differs from the usual data structures which can be generated by grammars. Instead of using the more typical tree-like data structure which branches out, the structure is more akin to a two dimensional grid, with an entry-point evaluated left to right. Performing an operation such as mutation would merely change the connectivity of the cells to each other.

Such grids are often fixed in size, this can contain bloating of the data structure. Because of its fixed size, it makes it ideal for use as a kernel in a GPU. This is covered in detail in a paper by Harding [22]. Cartesian Genetic Programming is also suited for FGPAs. Vasicek [23] mentions how a 30-40 times speed up can be reached using this hardware over its optimized software equivalent.

Finally, Weimer [24] outlines a method of automatically finding software patches, they were able to take advantage of mutlicore hardware, which has become more prevalent in recent years.

2.6 Modifications and Extensions

There are numerous variations that have emerged over the years. In this section, some of the more prominent and recent are covered to give the reader an idea of what is currently being used which differs from the more generic forms.

2.6.1 Compact Genetic Programming

The Compact Genetic Algorithm is an Estimation of Distribution Algorithm (EDA), also referred to as Population Model-Building Genetic Algorithms (PMBGA), an extension to the field of Evolutionary Computation. The basic idea is that rather than try to model the population in its entirety, a more simplistic model is used which is based on probabilities. As the simulation runs, individuals are then evaluated and the population as whole changes as now the distribution of the various probabilities has changed. This is the basis for extensions such as the Extended Compact Genetic Algorithm (ECGA). Such an approach is outlined by Sastry[25].

2.6.2 Embedded Cartesian Genetic Programming

Embedded Cartesian Genetic Programming (ECGP) extends Cartesian Genetic Programming by reusing partial solutions. These are called chromosomes. ECGP is the equivalent of putting blocks of code into functions or modules which can be called by other GP program code. Further details of this are given in Walker [26]

2.6.3 Probabilistic Incremental Program Evolution

Probabilistic Incremental Program Evolution combines probability vector coding of program instructions, population-based incremental learning, and tree-coded programs. PIPE is efficient when a short run time of the generated programs is desired. An example of this may be a program trying to find a way through a maze where it makes sense to penalize programs which took longer to find the end of the maze. Such is the example used in Salustowicz[27]

2.6.4 Linear Genetic Programming

Although the tree based variant is more common, Linear Genetic Programming (LGP) is another possible way to represent program genotypes.

The structure is similar to that of a simple machine code language where flow starts at the top and through the use of registers or conditionals, control flow moves up and down a la jump operations. One argument in favor of using LGP is that because there is no physical branching structure, there is less of a possiblity for large parts of the

code to become redundant or bloated as much. LGP, like TGP, also has parallelized implementations on modern PCs and video game consoles [28].

Wilson [29] did a comparison between LGP and Cartesian Genetic Programming. Differences between the two implementations are outlined. The most significant difference between them is each algorithm has a different means of restricting interconnectivity of nodes. The paper then does a benchmark between the two.

2.7 Potential Issues Arising Using Genetic Programming

Genetic Programming being a subset of machine learning, also is subject to a lot of the same common pitfalls such as overfitting. For example, when the fitness function just uses a small set of examples and the population become skewed towards providing good solutions which aren't very robust.

The generated code from individuals, given enough size can sometimes be difficult for a human to interpret. This may or may not be an issue however depending on the needs of the person using these programs. Code bloat is the accumulation of code which is unnecessary. An example of this would be in a tree representations which continued to branch out grow and grow, containing code making no difference to the overall fitness. If left unchecked, it can cause performance problems. Panait and Luke explain this problem in more detail and propose some remedies to to it [30]. A more recent paper [31] from 2009 suggests a more dynamic approach to dealing with it as well as providing a review of existing solutions.

It's also worth pointing out that code bloat isn't just always a small amount at each generation, as one might expect. Quadratic growth has been reported [32]. Whereas it is possible to use something like a compiler at a later stage to reduce the generated code/phenotype by optimizing it, code bloat accumulating as the GP is running is likely to cause performance issues if left unchecked. The have been some interesting means of reducing it. One proposal is to reduce bloat by giving preference to simpler solutions which tend to generalize well on unseen data, in accordance with Occam's Razor [33]. The solution in question was for a network intrusion detection system.

2.8 Summary and Conclusions

It seems evident from the variety of examples given that genetic programming has a wide range of applications, a lot of them having obvious practical benefit. It's also apparent that at least in some areas of application that the results are human competitive.

It also seems that the field has come along since its early roots and methodologies have become more formalized and literature around the theory has also grown and continues to grow.

We have also seen that GP, like other forms of evolutionary computation, is highly parallelizable and such implementations on modern hardware do exist and are in wide enough use.

Modifications and extensions to the archetypal GP do exist and have their own advantages and disadvantages.

No method is without some limitations or disadvantages and GP is no exception. We have seen how problems like code bloat can be a serious problem even on modern systems and what techniques are currently being used to mitigate its effects on performance.

Chapter 3

Approach

3.1 Key Early Decisions

3.1.1 Deciding on how to Evolve Agents

Having decided to use an evolutionary algorithm to evolve the agents, the first task was in deciding on the specifics of how to do this. Seeing as the desired end result was to be the equivalent to agent code, genetic programming seemed the most feasible way to do this. From there, the agent code would need to be evaluated and evolved towards better solutions. Running a game in its entirety, including displaying graphics and anything which was not needed for the agents to run during the evolutionary simulation would prove very costly in terms of performance and it was decided early on to try to avoid doing this if possible.

3.1.2 Coming up with a Work Plan

It was decided to have a high level plan consisting of four main phases. Each phase would build on work done previously.

Phase one would consist of researching, designing and implementing a very basic genetic algorithm and game to go with it. The main emphasis of this phase was getting up to speed with genetic programming and having a basic working prototype which ran through the command line. One classic problem in genetic programming which was chosen as a starting point is symbolic regression. That is, evolving mathematical

expressions which match a given equation. Programs are evaluated with a set of sample data and the results are compared with the results of the same data on the given equation. Programs which are closer in their numerical answers are favored over those which aren't. From there, a more interesting problem called the artificial ant problem was considered. This entails evolving code which controls the movement patterns for an ant on a 2D grid in search of food. It was found in Koza's book and thought to be a an ideal starting point for a game. Completing a project in a similar vein to this would subsequently end the first phase.

Phase two was to start with adding a graphical component to the game which was made. Running generated generated agents and periodically pausing the simulation to look at a text based representation worked but it left much to be desired. This work was done carefully, to ensure that the game could run from both the command line for the evolutionary simulation but then also allowing a graphical mode for people to actually play the game and also to see the agents behavior in action more clearly.

Phase three was to move from the previous turn (or discrete time) based game to more of a 'real time' game - that is more in line with modern games. The difference would be something akin from going from a turn based strategy game to a real time strategy game. Phase three would end with a game and some agents which demonstrates that evolving agents for a real time game is possible. It was anticipated that this might be quite different than work done in the previous phase. Large parts of new game logic may have to be implemented as well as some unique challenges associated with that would be encountered. So it was given it's own phase for this reason.

Phase four was created in mind for a more polished real time game, adding extra features to make it more like a typical game. 3D was considered an option at this stage but it wasn't deemed essential. What was more important was ensuring that the real time game and the agents which demonstrated that this could really be used in fully functional real time game. If time permitted, further polishing such as the addition of 3D graphics was then to be considered. Finally, during this phase results were to be collected and work on this report could begin.

Agile development methodologies were also considered in the planning stages. Feature driven development was decided on to be the best fit as it was noticed that an incremental addition of features would be required to get through each phase.

3.1.3 Technical Decisions

During planning and research a wide range of technologies were considered. Initially this entailed high quality open source games, game engines and various toolkits. Among these were both commercial and open source software. Unreal Tournament and a modification of Quake III from Stanford University which is used in testing agents in realistic scenarios called Urban Combat Testbed [34] were both considered. Each one of these technologies and games had their own unique set of advantages and disadvantages.

Given all the possibilities, it was important to consider the project goals, plan, time available and methodology when doing so. The project goals were carefully thought about and after some reflection they were expressed as a set of software requirements. These which would then aid with making key technical decisions.

Ultimately, it was decided at a high level, to start out small, with a working prototype first and then build on it incrementally. It's important to remember that in the early stages this project was very exploratory in nature and it was unclear how well the technique may perform computationally and in the context of games. It was believed that after there was a clear proof of concept in the form of a rapid prototype, more ambitious aims could be realized.

Given the high level project plan, the chosen agile methodology and the time frame, rapid prototyping was highly desired. In terms of deciding on a programming language, this automatically favored high level languages such as Python, Ruby or Lua.

There was also a need for a graphical component to demonstrate the games, and while many languages have capabilities for drawing graphics, the strong need for rapid prototyping was key again. So in addition to providing graphics capability, any libraries which accelerate development and abstract away low level functionality, particularly with games in mind would be ideal.

The author's experience has been with largely web development and desktop applications so it would be beneficial to use technologies which play to these strengths. The possibility of using mobile platforms, while it was very much possible, it was not really clear the benefit of spending time learning them would provide to the project and its goals. Having the software work cross platform was desired, but it wasn't necessarily a deal breaker for technology which turned out of be highly suited to other needs.

Lastly, perhaps one of the most important choices was choice of framework or

library for use with the actual evolutionary algorithm, genetic programming. The choice of going down this route, as opposed to coding this individually was to save on development time, to get up and running quickly and also to get the additional benefits of a good library. Additional features which provide added value with little added development time were something to look out for.

Ultimately, an evolutionary computation framework called DEAP (Distributed Evolutionary Algorithms in Python) was used [35]. The primary advantage of this framework is that it incorporates a lot of the standard building blocks of evolutionary computation, makes them easy to modify as well as having parallelism working out of the box. Two other genetic algorithms frameworks in Python were considered, namely Pygene [36] and Pyevolve [37]. Pygene, while very easy to use, was pretty bare bones in it's support for genetic programming. Pyevolve was more customizable in this regard. However, both frameworks left much to be desired in the way of getting parallelism working without having to write boilerplate code, so it was for this reason I decided to go with DEAP.

Python also has some mature and widely used libraries for creating games, notably Pygame [38] and Pyglet [39]. The Python language itself also ticked other boxes such as being high level and cross platform. So I decided to go continue with it throughout other aspects of the project such as simple scripts for setting up and running the simulations.

3.2 Design

3.2.1 Design of the Games

Using the feature driven development methodology, it was important to consider how implementing a particular game feature might impact the games from a practical point of view. It was important from an early stage to strike a balance between getting close to what people would expect from a typical game and distilling things down to important features which were valuable from purely an agent capability standpoint. For example, it's not unheard of in a wide variety of games across different genres to provide a degree of customization of game characters. While certain aspects of this can matter (namely non-aesthetic attributes), it was important not to invest a lot of

development time in this as it was not necessarily essential to providing an entertaining and interesting simulation.

When considering game mechanics, it was very important to consider how they might affect the gameplay overall and not just be added for the sake of making the game more interesting. Another important thing to consider was how one or more mechanics can be translated into some sort of heuristic function for use in evaluating an individuals overall fitness. This is not necessarily a difficult thing to do, but the reasoning behind thinking carefully about it will encourage thought about how that particular feature might translate into adding value to the simulation. For example, in a platform game, jumping adds a significant amount to the dynamics of the game, whereas the ability for the character to duck may provide some value in terms for both entertainment and the simulation, it's clear that jumping would be worth prioritizing.

The ability of the evolutionary simulation to converge on solutions is not to be ignored either. Lack of consideration for level design will lead to situations where the agents can't actually complete the level. This happened primarily early on in the experimenting with procedurally generated levels. There were cases where a platform couldn't be reached in a platformer or a maze was filled with dead ends.

Pathfinding Game

The pathfinding game started off with similar gameplay to that of the artificial ant simulation. It differs in several ways though. The end goal is not to collect all the food on the grid but rather, food sustains the agents and not eating leads for enough consecutive means they to die of starvation. The end goal was to reach a square which would then trigger a victory condition. Numerous additions were added such as different types of terrain which have their own unique movement/metabolism costs. The agents therefore must strike a balance between reaching the end goal and also finding food along the way. The final version of the game spans multiple maps and when graphics and gamplay were added it had similar feel to a genre of games often referred to as 'roguelike'.

2D Platformer

Moving a 2D grid towards a more 'real time' game, it was decided that a 2D platformer would provide an interesting dynamic and provide value in terms of both entertainment and exploring agent capability. 2D platformers are also widely recognized due to a number of popular commercial games like Super Mario Mario Brothers so it made for an ideal testbed for illustrating the potential and capabilities for the project for real time games.

3D Platformer

Keeping in mind that the fourth phase of planning entailed polishing and refining the real time game, the 3D platformer is the direct result of this. It started out with enhancing the existing code base for the 2D platformer. This included adding features common to platformers such as hazard tiles, collectables and basic enemies. The final step was to convert all of this to work in 3D. With the third dimension and 3D models added it does look a lot closer to a modern game and ultimately like the 2D platformer, the idea behind implementing it was to help illustrate the potential and capabilities of what is possible.

3.2.2 Design of the Agent Languages

It was decided from an early stage that whatever type of language was used that it ultimately would have to be translated into python code automatically, as that what the various games would be coded in. Initially, a lisp-like language was used for the agent behavior, particularly during the first phase of the project. This worked well, especially since DEAP makes it easy to specify a language like lisp for genetic programming because it is still very much the archetype for use in genetic programming.

However, it is also important to consider a potential target audience for this project, namely game developers. It's likely that not all game developers are intimately familiar with the lisp language, so even at an early stage the possibility of using other languages was also considered.

What was most important to consider in designing the agent languages was that it be as high level as possible. This would reduce the overhead associated with having to

read generated code as it would be more desirable to read through generated code at this level than at a lower level. Of course, it's entirely possible that someone may not really have a need for wanting to understand the generated code at all, but given the exploratory nature of this project it was considered important.

3.2.3 Design of the Evolutionary Framework

Having chosen a framework it was important to have a good sense for how to integrate it properly with the various games which were planned. Rather than take a top down approach and come up with a design for how the framework could be integrated with any game (such as explicitly designing an interface), a bottom up approach was decided on. It was decided to focus on a working genetic programming problem and then a very simple simulation with which would then be integrated.

The actual genetic programming problem for Symbolic Regression and the game was a basic version of what would become the Pathfinder game. It was then decided that once they were both working correctly to integrate the two together. A large top down approach for design and integrating everything is also not very likely to be in-keeping with agile methodologies either. As with rapid prototyping things can change quickly, it was preferable to avoid any over-engineering, especially at such an early stage.

With the approach which was chosen, it was to be expected that over time it would come to be that work on actual integration code would arise based on need. In accordance with feature driven development, it means that unless that this additional work was directly related to contributing to new features a user of the system could see - it was not given priority.

3.2.4 A Data Driven Approach

It was decided that during prototyping of all the games, great effort should be made to ensure that a lot of the attributes in the games could be customized easily and tuned using data (in the form of text files for example) rather than hard coding things into the games. This allowed for a great deal of experimentation and aided in speeding up the design process as well as the creation of levels.

Chapter 4

Implementation

4.1 Overview

The project can be seen as split up into three different parts, one for each game. Each of these are then divided up into two parts. One is the main entry point for the application and deals with setting everything up. This includes handling command line arguments, loading any levels, save states or agent code. The second part relates to the simulation. It contains everything needed to evolve the agents and uses the DEAP library. The simulation part can be thought of as a module which is invoked by the main module if the user of the system wants to evolve agents.

4.1.1 High Level Architecture

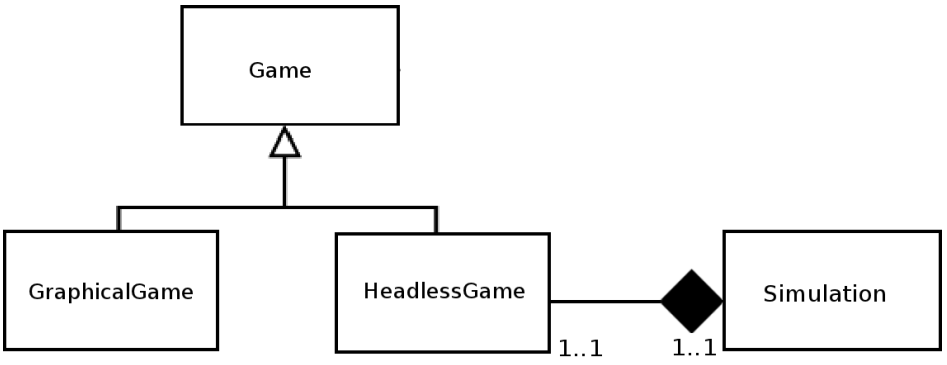

Figure 4.1: UML Class Diagram depicting the relationships for the main classes.

Figure 4.2: UML Class Diagram depicting how the simulation class is structured.

Keeping the Architecture Generic

During development of the first game a great amount of care was taken to ensure that any parts of the code specific to a game, however small, were to be strongly avoided in being included in part of the overall architecture. The creation of modules was seen as key means of doing this. They were kept generic and abstract enough with a mind to making it easier to reuse later. While it was a little extra work initially to do this during development of the first game, it became especially valuable later. It made it very easy to import relevant parts and only implementing exactly what was needed around this on a per game basis.

Polymorphism

As stated earlier in the approach chapter, it was desired to try to establish a clear separation of different things in the code depending on what was needed at run time. For example, rendering graphics for when people played the game and then not rendering any graphics for when the agents were being evaluated during evolutionary computation.

Polymorphism is an object orientated programming technique which made doing this a lot simpler. So in the stated example, a game object could inherit from a class which contains a lot of common information. Where differences emerge, they can have separate implementations. Something like drawing could be defined as a method but how the drawing actually gets done can be up to that particular class to implement. If the evolutionary algorithm was running, it could mean printing text out on the command line under certain conditions (like a debug mode). If a player wanted to actually play the game then this draw method would be drawing 2D or 3D graphics instead.

4.2 Integration of the Games with DEAP

DEAP allows developers to implement their own fitness function which takes an individual as a parameter and returns as fitness score. How the developer actually decides to do that is entirely up to them. In all the games, the individual is a game agent object, which has various attributes like the code associated with it and an a reference to a Game class. What the evaluate function does in the case of the games is actually start a new game by creating a new instance of that class. The parent game class has the behavior of simply calling the update and draw methods until it finishes. When it does finish it returns a score. This is then returned by the evaluate function as the fitness of that individual.

4.3 Game Implementations

4.3.1 Pathfinding Game Implementation

Game Logic

Agents traverse a map represented as a 2D grid. This was originally represented as text characters but then later with the addition of a graphics mode it can be displayed as tiled grid of textures. The game is played in a turn based manner, with each update corresponding to a particular move. With each step the agents use some of their stamina. Figure 4.3 shows an early version of the game.

This is dependent on what tile they end up standing on as different tiles incur different costs. Food tops up stamina. If an agents stamina reaches zero, the agent dies and the simulation stops. A long term goal of the agents is to reach the end square. Some squares are impassible, meaning agents have to navigate obstacles. A later enhancement meant that agents can use doors to travel between multiple maps, this is shown in Figure 4.4 which depicts the most recent recent version of the game.

Procedural Level Generation

Initially, levels were generated entirely randomly using the built in random library which uses mersenne twister for the actual pseudo-random number generator for level generation. When this was just two tile types, grass and concrete, the level was filled

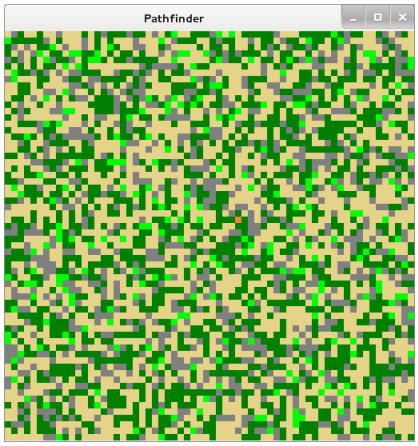

Figure 4.3: Early version of Pathfinder. No texturing yet. Food is represented as bright green, agent is orange. Two different terrain types are present: grass and concrete. Eating food yields a grass tile. Map is large because there is no support for multiple maps yet.

with grass with concrete tiles scattered throughout. The results of this technique varied considerably. Functionality was added around the basic random scattering of tiles. Probabilities could be given for the various tiles and the distribution of them all added up to one. For example if 0.5 was specified for grass, then half the total tiles in the map would be grass. If this number was kept constant then the only thing which would change throughout each level is the location of the actual tiles, not the amount of them. Various things like the amount of food per level could now be tweaked.The results of this looked very similar to Figure 4.3. This was an improvement but it still led to corner cases, for example food tiles trapped around concrete tiles which were inaccessible or worse, a maze of which couldn't actually be solved. A* Pathfinding was used to carve out a path from the starting position to the exit. Tiles were shuffled out of the way of this path. This ensured that there was always one solution to each level.

Fitness Function

Euclidean distance from the last recorded position of an agent to the end goal is used in conjunction with the agents metabolism. One divided by the remaining metabolism (plus a small amount) is multiplied with the distance when the simulation ends as a heuristic. Having this work across multiple maps involves summing the euclidean

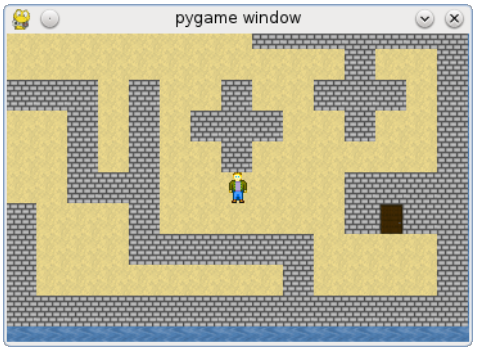

Figure 4.4: The most recent version of the game showing texturing as well as support for multiple maps. Character also does a walking animation when going from one tile to the other.

distances between doors. Using A* was considered for finding the ideal route, but wasn't used because the food locations were somewhat random and there was the possibility that the ideal path did not have enough food nearby for the agents. Agents got an additional bonus score for eating food. This helped distinguish those who happen to get close to the goal simply by wandering with those who were getting close to the goal but also on the lookout for food.

Web Based Level Editor

A HTML based level editor was developed in order to make constructing maps less time consuming. Maps had different tile types, this was originally represented with numbers in a text file. With the addition different terrain as well as various transition and corner tiles to go with it, looking up numbers for the various tiles became quite tedious. For larger maps as well it was hard to see what exactly things would look like so time was being spent running the game to check. This is where the level editor came in and it is depicted in Figure 4.5

The general idea was to use the DOM (Document Object Model) to create a 2D grid of images on a webpage. These can then be changed by click and drag actions with a cursor, almost in the same way a paint program works. Different tiles can be painted on the grid with the mouse, changing the images and giving visual feedback. Changing the tile to paint with is a matter of clicking sample tiles from a palette of possible tiles. Support for keyboard and mouse was also added, as well as an eyedropper (ability to

Figure 4.5: View of the web based map editor. Level data could be copied from and pasted in to the text box.

change current tile to paint with with one already on the grid). The application makes use of a lot of Javascript to do all of this, inspecting page elements and updating things accordingly. A text area was created for storing text relating to loading and saving. Loading in a map file is as simple as pasting in the text into the text area and clicking load to see it display. Clicking the save button will update the text area with whatever contents is currently being displayed.

4.3.2 2D Platformer Game Implementation

Game Logic

Whereas the previous game was turn based, update function of the 2D platformer has to continue to make changes to the state of the simulation even though the player or agent hasn't necessarily performed an action (due to things like gravity for example).

The solution to this was to first experiment with developing a simple 2D platforming game which is shown in Figure 4.6. In using the Pygame library to control the character, state information from the keyboard was polled on each game update. This is a typical way of doing it in Pygame (and in Pyglet and many other libraries as well). It was

then realized that for agent control, instructions in the agents programming language could then change this state accordingly. For example, an instruction to move right would change the state of the keyboard so that the right key was now pressed down.

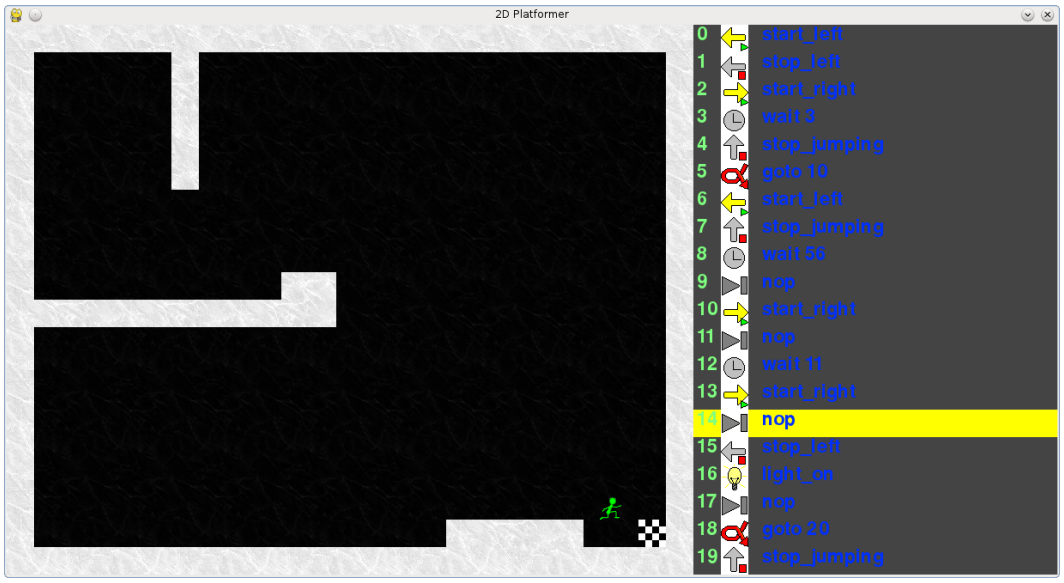

Figure 4.6: Screenshot of the 2D platformer running evolved agent code for that particular level. Note the redundant code on the right. All that is needed is to move right and jump if a collision occurs on the right.

For the actual headless mode though, there was no need to use the clock object provided with Pygame. This object is used to ensure that various methods like drawing to the screen are called at specific intervals, which is useful for displaying animations for example. Running without graphics though alleviates us of those concerns, and we can call the update function to the simulation sequentially without any delays.

The actual platformer game logic is not overly complicated (it is not integrated with a physics engine for example). The aim was to keep things simple and to try to avoid the need for anything other than commodity hardware. The aim of the game is exactly what one would expect from a typical platformer: go from one side of the level to the other, reach the goal and avoid and obstacles or hazards in the process.

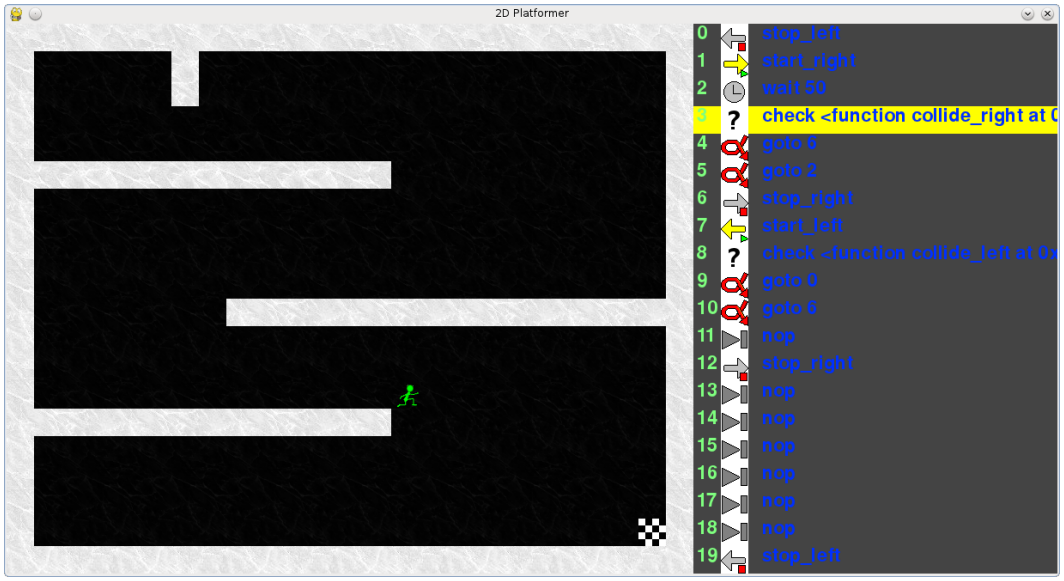

Figure 4.7: Another screenshot of evolved agent code running and completing a more complicated level.

Fitness Function

The fitness function in this case was a simple matter of calculating the euclidean distance from the character to the goal. If the agent dies the simulation is terminated with the last known position used. Added penalties were considered for dying but it turns out that distance was enough by itself to evolve agents which reached the goal. In the case where there are multiple ways to get to the goal, and as agents were reaching the goal (yielding a distance of zero), it was decided to add another factor which was the number of game updates to reach the goal. These were simply multiplied together.

There were many cases during evolution where agents are being evaluated in the simulation and they get stuck. Their code might be stuck in a loop walking and they would be up against an obstacle and not actually going anywhere. A time limit based on game updates was put on the simulation, as well as incurring a cost for agents which went over it. Later, some improvements were added such as if the agents position hasn't changed for a large amount of game updates then abort the simulation, assuming it will continue to be stuck and give it a high penalty. This improved performance but there still are a lot more cases to consider (like if the agent got stuck in a loop jumping or moving from side to side).

Live Display of Agent Code

During the transition to a real time game, a need arose to actually observe agent behavior in the level and also have a visual debugger of the agent language. In the previous game, being turn based, it's possible to examine things carefully state by state in one's own time, with the generated agent code in a text file open going through it step by step to examine it.

Whereas with the real time nature of this platformer, the agent can change its behavior just as quickly as a human player would and levels were replayed just to see the exact behavior multiple times. This was also during a time when I had decided to work on a new language for the agents which resembled the basic programming language. It was of great importance, primarily to aid with tracking down bugs.

The actual display itself was itself an added front end layer on top of the interpreter which was written for the basic-like language in python. Properties such as the instruction pointer were used to highlight the current instruction and the various instructions were all given icons and line numbers. This is all auto-generated from the given input program and were for added clarity. It helped a great deal in troubleshooting during development. Bugs could be in a number of places in the pipeline while it was still being developed and seeing the observed behavior alongside the actual program running in real time made it faster to track down and fix things.

4.3.3 3D Platformer Game Implementation

Game Logic

The 3D platformer had the game set of goals in mind as the 2D one only with a mind to making the simulation world more like a modern game rather than a retro one. For this enhancement, a great deal of work was actually needed. The display code, game logic like the collision detection and even the agent language all required large changes of some kind to ensure that the transition to 3D was a success. Code was reused where possible, and given the generic architecture, it was possible to get around to adding changes specific to the 3D game rather than removing chunks of 2D game code which wasn't needed for example.

A prototype for the 3D platformer was developed, which used the same 2D grid

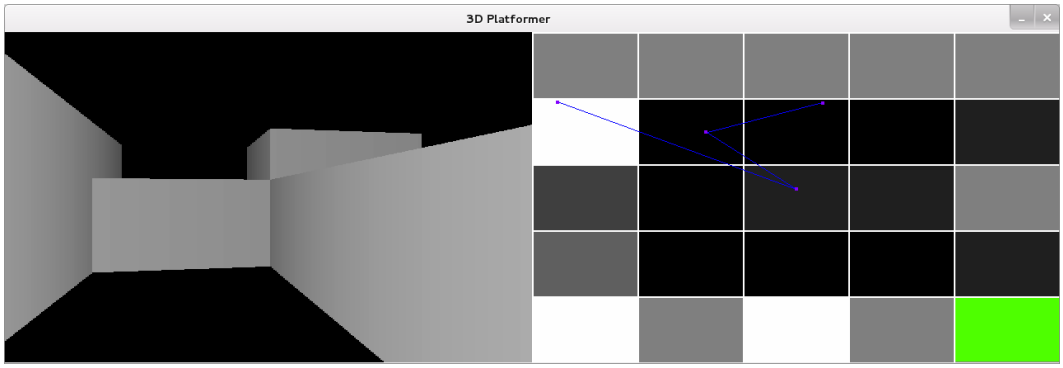

Figure 4.8: A screenshot of a prototype of the 3D game. Display on the left is for the height fields and also to track where the agent has moved. The green square is the goal.

representation only it became a height field. This is shown in 4.8. Instead of simply being passable block or not, as was the case for the 2D platformer, a range could be given and this was then rendered as a cubeboid shape which was of a certain height. This would work but it would place limits on the kinds of maps which could be created. For example, it wouldn't be possible to have rooms with ceilings or windows in rooms. Alternative methods of representing and drawing a 3D scene were then sought after.

The height field representation was modified to allow for multiple height fields layered on top of each other. However it became tricky to actually create maps in this way as it was not very intuitive as say editing a single 2D grid anymore, there now were multiple grids of varying cell heights stacked on top of each other. It was eventually scrapped in favor of a cube based representation. This added greater flexibility in terms of what could be created over a single height field. It seemed like a logical progression after multiple heightmaps to standardize the range at which the height could grow. Cubes were easy to grasp in terms of a basic unit for 3D levels. With a standard size it became easier to build things.

The part which took the most was actually the display related code. Rendering things efficiently, turning on backface culling was just a starting point, dividing what was displayed into chunks which loaded dynamically became preferable for large worlds. Model loaders, animations for the models and stylized rendering of the blocks (based on their neighbors) also took up some time.

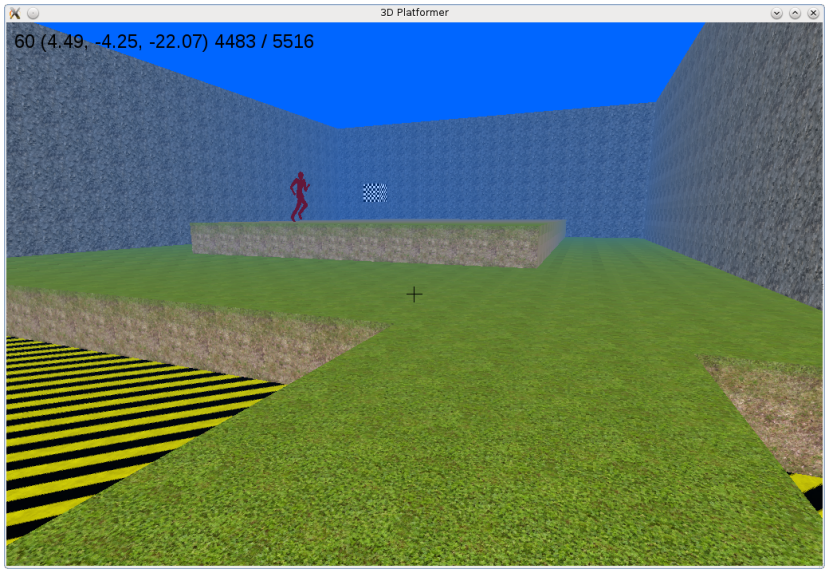

Figure 4.9: A screenshot of the finished 3D platformer game. Notice the checkered cube in the background, this is the goal. The enemy is patrolling in front of, as if to guard it. There also pits which the player can fall into.

Fitness Function

A lot of the lessons surrounding the heuristic and convergence had already been learned on the 2D version so it was a matter of keeping them in mind while developing the fitness fuction for the 3D version. With The fitness function turned out pretty much the same as the 2D platformer, with the added dimension included in the euclidean distance calculation and also checking whether the agent has gotten stuck.

In-Game Level Editor

The 2D web based level editor wasn't really good enough for editing large levels in three dimensions, nor does text files with a grid representation do much better. Converting the simple text file with numbered blocks into a format something a popular 3D program like blender could read was considered. Implementing a very basic 3D editor was then also considered. Then a moment of realization came when I was testing the game, I already had a basic 3D editor, the game itself. Using the game itself made it much easier to edit the levels with minimal cost in terms of development time. The main task was figuring out where the nearest block in 3D space was relative to center

of the screen, also keeping in mind a threshold value for distance(avoiding accidentally adding a block far away and not noticing it).

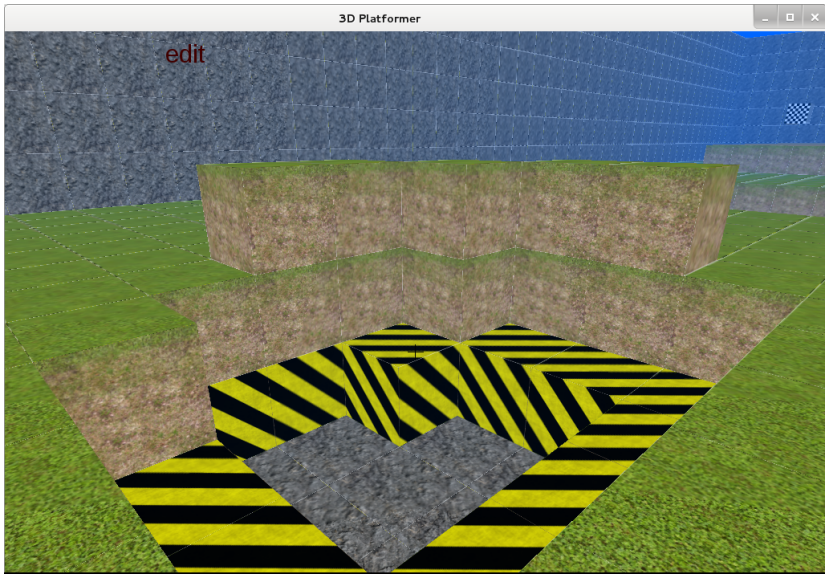

Figure 4.10: A screenshot of in-game level editor for the 3D platformer. Click right and left clicking the screen adds and remove blocks, allowing for the creation of 3D level structures.

For the in-game level editor, it was important that there was a clear distinction between playing a level and editing one (so that players don't accidentally edit the level during play). So the concept of 'edit mode' was implemented, allowing the player to do things not normally allowed - like fly around the level (making it easier to build levels also).

Procedural Level Generation

During the creation of the levels the possibility of procedurally generating the 3D levels was explored. Ultimately, running a few tests on these large levels took a much longer time to converge, making them less desirable for use in the actual final levels.

However the functionality is still there and can be used. Vast hills and valleys could be created as shown in 4.11. These were not used in the final levels. However there was high level functions to create boundary walls and floors using a few parameters. These were used for the smaller levels, this is shown in Figure 4.8. Another example is

The procedural generation code saves on time and makes creating levels less tedious.

Figure 4.11: A screenshot of a procedural level. Showing hills of various sizes with a base grass floor.

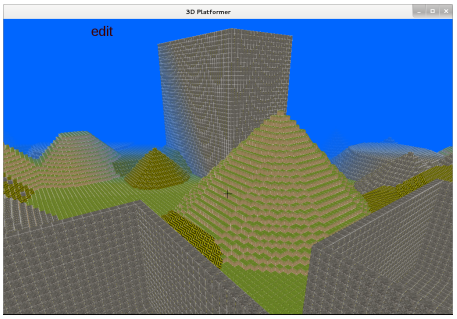

Figure 4.12: A screenshot of a procedural level. In this screenshot procedurally created rooms are visible.

4.4 Agent Language Implementations

Whenever an evolutionary simulation was completed, a way of persisting/saving these agents was needed for later use to run in the game in graphics mode to actually observe agent behavior. Initially this was just using serialization of python objects. While this worked fine, for actually inspecting the agent code and showing it to others, a clearer more readable output was desired.

The basic principle is to translate the generated agent python code into a simple text file format which can then be read in and translated back into python code. A key factor was making sure that this text file format was concise, easy to read and well formatted. Simply dumping the resulting Python code to XML or JSON or similar format would not make it concise.

Given that the programs in both languages already had structure, it was decided to make the output file represent this structure in a meaningful way. The lisp-like language would have brackets and nesting for expressions and the basic like language as a sequence of instructions with new lines to separate them out. An overview of the output pipeline is given in Figure 4.13

With these output formats decided on, it was then necessary to write a parser and lexical analyzer for each language format, so that it could be translated back into

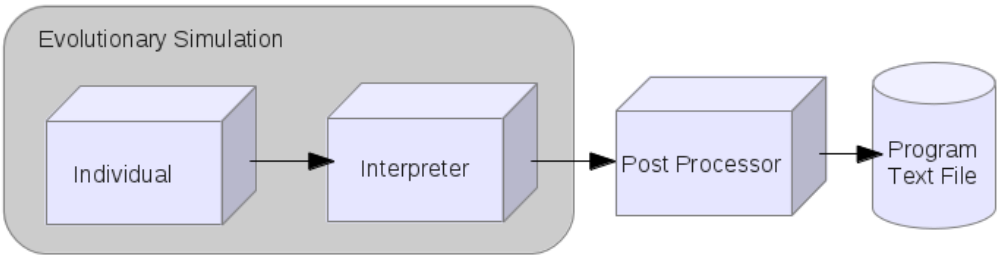

Figure 4.13: Overview of the language output pipeline.

python code and run inside a graphical game. This became the language input pipeline and a high level view of it is shown in Figure 4.14

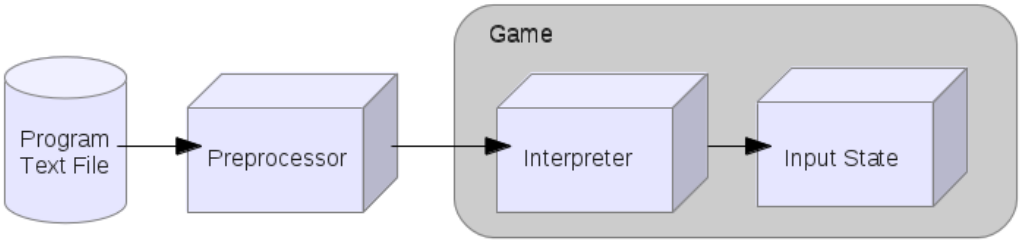

Figure 4.14: Overview of the language input pipeline.

4.4.1 Lisp-like Language Implementation

As stated in the background and state of the art chapter, a branching tree-like data structure is the archetype of genetic programming. Running and evaluating programs is then a matter of traversing this tree (with budding off into different branches based on conditional statements). An additional factor in genetic programming in general is the ease of use by which code itself can be treated as data.

The lisp language is well suited for both this kind of data structure and its ability to pass code around as data. It is therefore not surprising then to see it in widespread use in GP. DEAP has a branching tree data structure built in for genetic programming. One can specify methods as nodes and terminals. These are python objects. Nodes provide branching functionality by allowing more than one parameter to be passed in. Ultimately, in a typical program, it results in a tree of nodes with terminals at the end, running the program results in control flow starting in a root node and finishing in one

of the terminals.

```
(prog3 move_forward
       (if food_ahead
           move_forward
           turn_left)
       move_forward)
```

Figure 4.15: Generated program code for the Pathfinding game. The resulting behavior is for the agent to move forward in an anti-clockwise manner, searching for food ahead. If it finds it, it moves towards it, otherwise it keeps searching.

A lisp-like language was then constructed which was built on this, in such a way that a subset of keywords and functions from the language as seen in Figure 4.15. Prog2 is from the Lisp language, executing two items passed to two it in sequence. 'If' is also from Lisp, evaluating the conditional and then executing either the second or third item passed to it depending on whether it is true or not. Ultimately this all translated into a system of nodes and terminals.

4.4.2 Basic-like Language Implementation

The basic-like language, although simpler in appearance and simpler to parse out, was actually trickier to implement. An example of some code from the language is given in Figure 4.16. A simple sequence of actions could be modeled as a chain of nodes which have a terminal at the last instruction. In other words, a tree with no other branches budding off it. However, a key instruction in the basic language is the goto statement. This allows for jumping ahead or backward in the execution of the program by altering which instruction is to be next. With this instruction, it's possible to split code up into different blocks as well as have certain behavior repeat continuously (perhaps until a certain condition is met). However traversing a tree from the root to one of the leaf nodes without turning back does not allow for cycles. A graph would allow this but the underlying implementation was based on trees and didn't have support for graphs.

To get around this problem, inheriting from the base genetic programming object and then re-implementing the parsing code to work with graphs was certainly possible and was considered. However this seemed unnecessarily complicated. A more elegant solution emerged, using the genetic algorithms code as a base for implementing this

```
0 stop_left
1 start_right
2 wait 50
3 check collide_right
4 goto 6
5 goto 2
6 stop_right
7 start_left
8 check collide_left
9 goto 0
10 goto 6
```

Figure 4.16: Generated program code for the 2D platformer. Redundant instructions removed for clarity. The resulting behavior is that the agent moves until it encounters an obstacle and then moves in the opposite direction.

custom language. In the genetic algorithms module, all data for representing the phenotype is represented as a list. This typically represents solutions to problems. In the traveling salesman problem, a solution is the order in which the various cities are to be visited for example. In the case of representing a basic language program, this was a list of instructions. Instructions were kept simply as python objects with basic properties with a mind to enhancing them later.

Evaluating an agent in terms of fitness now meant that the instructions would be parsed out from the list and run in an interpreter. The interpreter acted as an interface for the particular game. It kept track of important details such as the current instruction, the state of any given inputs (whether the agent wants to move left or jump for example). It also had a very simple form of memory which could be wrote to and read from. Figure 4.17 has a sample of code which demonstrates it's use.

An additional instruction called the 'check' instruction is added. It takes a single conditional function as a parameter (something which returns true or false). If the condition is true, it executes the next instruction, if the condition is false, it skips over the next instruction. This is shown in both Figure 4.16 and Figure 4.17.

The use of storing conditional variables have the potential to be used to keep track of things. For example in Figure 4.17, the agent would jump if it collides with something on its right but not only that, this specific event is captured as a true or false statement which could be queried in other parts of the code, in this case it's read from to determine whether the agent should jump or not.

```
0 start_right
1 check light0
2 start_jumping
3 wait 50
4 stop_jumping
5 check collide_right
6 light_on light0
7 light_off light0
8 goto 1
```

Figure 4.17: Sample code showing the ability for the program to store information. Programs can use 'lights' to store data, simple boolean variables which can be set on and off and read from. In this case the program is storing whether or not it is colliding with something to the right or not.

Some adjustments were needed to make the basic-like language work in 3D, a sample of this is given in Figure . The keys up and down had new meaning in 3D. Related instructions were remapped to moving forwards and backwards in the world, on the Z-axis, without triggering any jumping. Instructions to rotate the camera as well as checking for collisions in the new dimension were also added.

```
0 start_up
1 rotate_x 15
2 start_right
3 wait 25
4 stop_right
5 wait 100
6 start_left
7 wait 200
8 start_jumping
9 wait 20
10 stop_jumping
11 rotate_x -15
```

Figure 4.18: Evolved agent code for a completed level for the 3D platformer.

Chapter 5

Evaluation

5.1 Methodology

Both the various games and their evolved agents were assessed in terms of their overall quality as well as feasibility and effectiveness for demonstrating the capabilities of the approach mentioned in this project. Various levels in each of the games were created for both human players and agents to play in order to do this.

The actual evaluation itself was done incrementally and experimentally as the games were developed in this way. In-keeping with the general idea of feature driven development methodology, this was usually done after specific features were implemented.

5.2 Assessment of the Generated Agents

5.2.1 Quality of Agent Code

Code Legibility

The question of code legibility is an important one if the generated code is to be used or modified at a later time by a developer. This may or may not be a factor depending on the application. Remembering the analogy of procedurally generated graphical content, sometimes it is desired for a professional artist to manually tweak the content after it is generated. So it is entirely possible that an expert may want to tweak the results generated. With this in mind, the best individuals from evolutionary

simulations of each of the levels of each of the games and assessed. This was done in careful consideration with the possible drawbacks mentioned in the background and state of the art, code bloat in particular.

Game Playing Ability

Samples of the various generated agents were taken and their behavior was observed while playing the games and levels in question they were evaluated on during their evolutionary simulation. The graphical versions of all the games have an ability to load in program code and run it as if a player was playing. Samples of each of the best agents from each of the levels were each loaded in to the games in this way and their behavior was carefully observed and notes were taken of anything which may be of interest later.

5.2.2 Performance

Convergence

As with any evolutionary algorithm, convergence of the individuals of a desired solution is something important to consider. Various factors such as the selection method for individuals and the implementation fitness function are all something to carefully consider. This can be very much trail and error. Ultimately, experiments were done to see what could be done to improve convergence on solutions. Variations in level design and fitness functions were considered. Any noteworthy observations in actually running and getting the solutions to converge are also worth reporting on during this and it was kept in mind to record these throughout the project.

Generated Agent Performance at Run-Time

A benchmark was done in order to see if the generated agents had any noticeable impact on the actual performance of the games themselves while they were running. A sample of agents of different levels of fitness were taken in order to do this.

5.2.3 Robustness

Having the same agent play well across multiple levels is something to consider. Having agents evolve on just one level runs the risk of them 'over fitting' to that level and having poor performance on other levels. Elements of randomness to the levels and assessment across multiple levels was considered to address this issue.

5.3 Assessment of the Games

It's important to ensure the games were playable and could actually be completed by both a human player as well as the evolved agents. Careful testing of each level of each of the games was done by both human players and the generated agents. Level editors assisted greatly in speeding up this process as a great deal of experimentation was done regarding level difficulty. It was also important to make sure the games themselves weren't completely trivial as well. Simple levels were first chosen both to make sure human players got used to playing and also to assess the feasibility of evolving agents in this way. This was to show proof of concept. The complexity of the levels then increased to demonstrate that these games can be challenging to players and that the agents could also complete the more challenging levels.

Chapter 6

Conclusions

6.1 Results

6.1.1 Resulting Agents

Given sufficient time to evolve, agents completed the levels of all the various games, so in that sense, the project was a success.

Resulting Behavior

Watching the resulting agents play, their behavior appears intelligent. Agents avoid obstacles as well as move in the direction of the goal they have to reach.

In the 2D games, agent behavior could very well be mistaken for human behavior. In the 3D games, goals are completed but they don't always have the same human-like behavior. While the agents still reach the goal, some aspects of their behavior don't match up closely with how a human might complete the level. Often the resulting agents don't look at the goal they were trying to reach while walking towards it.

This is because of how the agent is able to navigate the environment. The controls are standard to first person shooter games. It's possible to strafe and walk independently to where the character is looking. Since the agents weren't getting any kind of incentive to look at the goal while they approached it, very often they didn't. A simple way to prevent this would be to restrict the camera control code to always look straight ahead in the direction the agent is moving. Another less restrictive way of doing this

would be to add some incentive for the agent to look at the general direction of the goal by awarding an improvement in fitness for doing so.

These methods only need to be applied if making the agents behave more human-like is actually desired. For example, if one was more concerned with using the agents for testing to see whether or not difficult levels could be completed then ensuring resulting behavior was human-like would not be a factor at all.

Resulting Code Legibility

As was expected, a certain level of redundant code was found in the generated agents. This varied depending on numerous parameters. The main ones being the difficulty of the level and the amount of subsequent generations the evolutionary algorithms ran for. Some attempts were made to alleviate this. The primary and most simple way of doing this was to place hard limits of how much the code could actually grow. This at least stopped the problem from getting out of hand. However, it's also worth suggesting that one must not constrict too much. For example, if a level is particularly complicated, placing a really harsh limit on the amount of growth can lead to situations where it is actually impossible for the agents to converge on a solution.

Overall the resulting code was not particularly legible, even with size and growth limits. It's not trivial to factor in some sort of legibility heuristic into the fitness function. Removing instructions which appear to do nothing as the simulation runs can have unintended consequences. Sometimes redundant code can become useful later and it's hard to know what won't become useful at a later stage and what will. With the placement of limits at least, it was possible to understand what the code actually does, after careful analysis. A first stage in analyzing the code was to look for any instructions which are clearly redundant and remove them. This could actually be automated as a post-processing stage. Again, going beyond removing the obvious redundant instructions in an automated way is tricky and definitely not trivial. It would take considerable time and effort to develop. It's worth pointing out that ensuring all generated code was perfectly legible to a human was not an explicit goal of the project either but rather to see if the generated agents were up to the task of completing the various levels.

Robustness

It was desired to have agents which didn't have behavior which could be considered rote learning or 'over fitting' to a particular level. It was observed in a wide number of samples that resulting agent code had specific instructions to move in certain ways which were specific to a particular level. It was hoped that the actual instructions pertaining to sensing would be used more in the levels. This was not the case in a lot of samples taken. While disappointing upon discovery, steps were eventually taken to try to reduce this effect.

In order to do this, a certain level of randomness was added to the levels. For example in the platformer games it could mean changing the locations of platforms or enemies. Evaluating the agents across more than one level also proved useful. It allowed for them to generalize concepts like if an agent bumps into something it's a good idea to try to jump or change direction. It's important to note that while these are simplistic concepts to a developer or a gamer, the agents have to learn these concepts without any guidance. It was shown that with a subset of levels, general strategies like this could emerge. This at least hints that it is possible to come up with agents which can perform well on unseen levels, given a sufficient number of different levels with elements of randomness added to them.

6.1.2 Resulting Performance

Convergence Rate

The rate of convergence wasn't a primary concern for the project, just that convergence was in fact possible and achievable in a reasonable period of time. That was more important. The reasoning for this is that it's entirely possible to do a computationally heavy evolution of the agents prior to shipping a game or even have them as downloadable content. With this in mind, some attention was given to convergence at the beginning but it became less important once it was shown to be possible to evolve agents which completed the levels in a reasonable period of time. It's also worth pointing out that while many different factors from tweaking various values all the way to refining aspects of the fitness function could have been possible it's important to remember both the project goals and the timeframe.

Initially, tweaking of various parameters of the evolutionary simulation began with the pathfinder game and then the 2D platformer. The 3D platfomer used the same parameters as the 2D one. The levels were briefly play tested first to ensure that they could in fact be completed by a human at least and then left to see if a solution could be evolved. A population size of 500 was used with tournament selection and a mutation rate of 2 percent. The actual mutation code was simple, it just changed one instruction in some way. Cross-over was simple, just one point was used to swap code between two agents. For more difficult levels the population size was increased to 1,000.

Ultimately, how long it took for a solution to emerge for a given level greatly depended on the actual complexity and difficulty of the level itself. Trivial levels of just walking from one side of the level to the other with little to no obstacles took less than a few minutes. Whereas a larger level with hazards and more obstacles took anywhere from a few hours to even a few days. Added randomness and multiple levels multiplied the time needed to converge.

In the absence of any of the issues encountered, such as allowing enough game updates during evaluation to complete the levels and making sure these levels could be completed, convergence on solutions to the level was possible.

Resulting Run-Time Agent Performance

Assessing the performance of the agents started with various agents being evaluated of different levels of fitness. After running a few samples it was clear that very little impact in terms of performance was observed, regardless of what code was running. Randomly generated individuals of large sizes were also tried but there was negligible difference in terms of computational performance at run-time. Remembering that all the graphical versions of the games have an option for loading in agent code and playing it in a simulation, it was also decided to try out hand-coded infinite loops (multiple goto statements in agent code repeating forever were tried). Again, no noticeable impact in performance was observed.

This was due to the fact that for the update loop of all the games, a single instruction was read and executed by the interpreter built into the games. This means that even a large amount of code or repeating the same instructions will have little impact on the game running smoothly.

Why is this of a concern? It was important to test such things during development because a single agent which is able to 'lock up' and get caught in a loop using lots of computational resources is not only detrimental to the gameplay experience, but also if this kind of thing happens during the evolutionary simulation it means that it will never progress if such an agent is encountered. Such a bug may not even be with the language specification itself but the implementation of it somewhere, for example all that is required is for some function or method to never return and the whole thing can 'lock up' under certain circumstances. Although this particular issue was never encountered during the project, great care was taken to avoid it from happening during development.

6.1.3 Resulting Finished Games

The final versions of the three games are comparable in terms of their gameplay and features to that of games which are commercially available. Obviously given the time constraints of the project they could not have as high production values or polish to them as their commercial counterparts but such things were not essential to achieve the project goals anyway. Some effort was however made to at least make them somewhat visually appealing which has been shown in the various screenshots of the games.

What is clear is that the games themselves have enough gameplay features and mechanics to be considered part of a particular known genre and leave little doubt as to whether or not they could be considered games. The pathfinding game can be considered similar in its gameplay to what is often termed a 'rogue-like' (this genre gets its name based on a text based exploration game called Rogue). The platformers have the kind of things one would expect in a typical platform game, platforms, pits, enemies and so on. This genre is also very well established at this stage so they do make for a useful testbed in terms of applicability of the approach taken in this project.

6.2 Criticisms

6.2.1 Level of Agent Intelligence

The resulting agents code is not that large, limited deliberately in its growth in a lot of cases based to aid with legibility in mind for later examination. So it is important to ask if this can really be considered intelligent behavior or not.

The author means intelligence in the sense of game artificial intelligence, which is not necessarily perfectly aligned with the field of artificial intelligence in computer science. Games usually fall under the category of entertainment. The pursuit of creating artificial players for games isn't necessarily a scientific exercise in mimicking how human intelligence works. There is more of a concern with getting the outward or surface level behavior realistic or believable enough for players to have an entertaining gaming experience. In this respect, simple state machines or giving the artificial players global knowledge ordinary players don't have is acceptable in order to get the game to ship.

The resulting agents don't have state machine logic or global knowledge at all. Global knowledge in the case of the games created for this project would be knowing exactly where the goal was in each level. Yet the evolved agents are able to navigate obstacles and reach the goal. Observed surface level behavior shows agents completing the levels so it meets the requirement for games. To a certain degree, they have internalized or learned the layout of the levels to do this, but in cases where elements of randomness or multiple levels is used they do make more use of their sensing systems. With further time and effort, it would be possible to add to the sensing system and run even more levels with more randomness in them to get even more robust and intelligent agents.

The resulting agents could be used in place of human created scripts for artificial players in games. This was the intended goal all along. Given sufficient time to evolve, they exhibit behavior which appears intelligent. This is not the same as actually attempting to create something which has the goal of behaving like an intelligent system by modeling the internals of an intelligent system. An example of doing this would be to create an artificial neural network.

6.2.2 Size of the Levels

The levels chosen for evaluating the levels weren't very large, taking somewhere from about thirty seconds to a minute to complete. It was felt that with larger levels, it might demonstrate greater capabilities of the agents. This definitely is worth investigating and it may very well in fact be true. Ultimately, the reason for keeping the levels brief was due to the amount of time it took for solutions to converge. As level complexity and size increased, the maximum allotted game updates for agents to find a solution during their evaluation also had to be increased. A level twice as big may take twice as long to converge (depending on layout). Getting the number of maximum allotted game updates right was also very important. If this number was too short, it would be impossible for agents to converge at all and if it was left too high then a lot of computational resources were wasted.

In order to ensure agents could be evolved in a timely manner, level sizes were then kept from getting too big. Getting the number of maximum allotted game updates per agent right was also being experimented with constantly. Larger levels brought greater uncertainty for the most efficient number. Remember that the overall, total number of game updates during the evolutionary simulation is multiplied on a per agent basis, so it has considerable impact on performance.

6.2.3 Language Design

The question of whether or not languages like the ones used in this project were a good choice is important to ask. A popular or more well-known language subset and syntax (like the C language for example) could have been chosen instead. This is certainly a possibility and there are numerous frameworks available which aid in doing this. Python was chosen as the language for this project and DEAP was then chosen as the framework, for reasons outlined in the design chapter. While it was possible to explore other languages and frameworks, for practical purposes and given the incremental, feature driven development approach, it made sense to reuse existing code. The languages then used for genetic programming had to stem from this. Of course, it was still possible even then to implement something quite different from the languages which were used in genetic programming for this project. The lisp-like language was chosen because it integrates very well into DEAP. Also as stated in the background

and state of the art chapter, this kind of branching style representation for a program in genetic programming is the archetype for genetic programming. It was important to implement and get right for those reasons. The basic-like like language was chosen because Linear Genetic Programming wanted to be explored and also because it would be a nice contrast to implement two different languages and variations in GP. It was also sought to have a more legible structure - rather than having lots of branches and nested statements, code could be read from top to bottom. It's also important to remember that because of the approach was based on feature driven development, even the language features were designed by feature. The likes of the if and check statements were created with the need for conditional statements to allow for different behavior of the agents depending on the level. Same for iterative statements like the goto, a need arose for a feature where the agents to be able to repeat certain actions without having to duplicate code - making it easier to read.

6.3 Future Work

The results have been encouraging and suggest further potential research. There is quite a lot of areas which could be expanded. This section will outline some of those which are more pertinent.

6.3.1 Co-Evolution

DEAP supports both cooperative and competitive co-evolution out of the box. For this project this would entail multiple agents being evaluated in the same level. In cooperative co-evolution both individuals performance would rely on the other agents they are playing with as well. In order for this to be meaningful, the agents have to be able to interact with one another in some form. Having the game handle colliding agents and expanding the sensing system to make agents aware of agents would help achieve this. Inter-agent communication may also be of use here, just like communication helps groups co-ordinate with one another in real life as well as virtual worlds in order to reach a common goal.

Competitive co-evolution might not require communication between agents although the ability to sense other agents would be useful. Interesting behavior might be ob-

served like one agent physically pushing the other out of the way to reach the goal first or at least get a better score than the other agent. Karl Sims Creatures [40] has some very interesting examples stemming from competitive co-evolution and it would be interesting to see what kind of behavior would evolve from competitive co-evolution in the various games which have been created in this project.

6.3.2 Different Games

There are many more games out there than the ones which are quite different to the ones created in this project. It would be worth trying the same technique out on different games to see what kind of results could be achieved. Different games would present different challenges, such as coming up with a good fitness function for that particular type of game, defining programming languages which work well for that type of game and so on.

The basic building blocks are there for both 2D and 3D games. Depending on the game, the agents may need more sensory information or perhaps could get by with no sensing system at all. No sensing system would be applicable in games which have certain ways of solving them, such as a deterministic algorithm - if followed will guarantee a solution. Actual games which are like this could be certain kinds of maze or puzzle games. A more detailed sensory system would likely be needed for some kind of game involving shooting. Currently, the sensing system is very local (in other words - agents are able to determine if there is food nearby or if they're bumping into something but don't have a detailed vision system). A more detailed vision system would be needed to sensing enemies from afar and targeting them, that is assuming there is a cost involved for shooting (like a limited amount of ammunition).

6.3.3 More Robust Agents

While efforts were made to generate agents which could work across multiple levels and tolerate a certain degree of randomness, improvements could still be made in this area. This would be especially important if agents had to know general principles surrounding levels - in the case where procedurally generated levels were being used for example. If this is not a concern it's entirely possible that the agents could just

learn across all the levels which were created for a given game and eventually come to master them given enough time to evolve and that would be enough.

If discovering general principles behind given levels are of a concern then more robust agents are definitely required. In testing this, agents would be required to play different levels with the hope that they would come to use these general principles also, just like a good player would. It's also not unheard of for players to find imbalances or bugs in the game which can be exploited. Agents can be used to explore the possibilities and do so faster than human testers.

Actually generating these more robust agents is simply a matter of allowing for more time for the evolution to work over more levels and incorporating a greater level of randomness into each level. This seems quite feasible given more computational power or time.

6.3.4 Integration with Games, Game Engines and Toolkits

It's quite possible to integrate the approach taken in this project with the various games, game engines and toolkits both open source and commercial. Research on this possibility was done very early on during the project. The main reason these avenues were not pursued was simply due to time constraints.

It takes time to learn and understand the various technologies, on top of that given commodity hardware was being used, a headless or graphic free mode for the evolutionary simulations was desired. It's quite possible to get this to work for various games, as a lot of popular PC games for example do have server side software for use in multiplayer games. Such software often has client code integrated without the graphical overhead to validate the actions performed by the connected clients. The option of extracting/modifying this code was considered, but modern games are quite complicated and the effort in actually doing this was more than initially anticipated.

For example, in the Quake III engine, time was spent examining its code base. It actually contains a custom made virtual machine which evaluates all the client side actions from players. Going through that code base, deciding what is and isn't needed was proving to be quite time consuming. It's unclear how such extensive modifications would be in-keeping with the project goals. It was also unclear exactly how long determining all the important parts, modifying and/or extracting them would take. So

simpler, less time consuming methods which could be assured to be completed in the timeframe were subsequently pursued.

However, there is nothing preventing the technique presented in this project from being implemented in other games, given sufficient time. The results of which would be quite important in order to make this approach and the benefits from using it more widely available and accessible to others.

6.3.5 Evolving Agents using Human Players

As part of the evaluation phase, human players could play with the agents, likely in a competitive setting. Fitness could then be then determined by the agents score relative to a player's score. The main reason this method was not pursued was the amount of time it would take for the agents to actually learn given that human players had to play alongside them. It might take thousands of plays of a level with a human for agents to develop skill against a human player. This would likely be very unfeasible, except maybe in an online setting if sufficient volunteers could be found. There is also a hybrid approach, agents can co-evolve competitively or against an agent which has been developed by humans. After a number of generations, the agents get to play against a human player and this will influence their fitness. Co-evolving competitively against human created artificial players is also valuable for determining if there are any flaws in the human created agents' behavior which may be exploited by players (both human and artificial).

Appendix

Acronyms

GA Genetic Algorithm

GP Genetic Programming

EDA Estimation of Distribution Algorithm

ECGA Extended Compact Genetic Algorithm

PIPE Probabilistic Incremental Program Evolution

LGP Linear Genetic Programming

ECGP Embedded Cartesian Genetic Programming

Bibliography

[1] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[2] D. B. Fogel. Nils barricelli - artificial life, coevolution, self-adaptation. *Comp. Intell. Mag.*, 1(1):41–45, November 2006.

[3] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[4] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[5] Robert I. Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, September 2010.

[6] Douglas A. Augusto, Helio J.C. Barbosa, Andre M.S. Barreto, and Heder S. Bernardino. A new approach for generating numerical constants in grammatical evolution. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 193–194, New York, NY, USA, 2011. ACM.

[7] M. García-Arnau, D. Manrique, J. Ríos, and A. Rodríguez-Patón. Initialization method for grammar-guided genetic programming. *Know.-Based Syst.*, 20(2):127–133, March 2007.

[8] W.P. Worzel, J. Yu, A.A. Almal, and A.M. Chinnaiyan. Applications of genetic programming in cancer research. *The International Journal of Biochemistry & Cell Biology*, 41(2):405–413, 2009.

[9] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic programming IV: Routine human-competitive machine intelligence*. Springer-Verlag New York Inc, 2005.

[10] A. Elyasaf, A. Hauptman, and M. Sipper. Ga-freecell: Evolving solvers for the game of freecell. 2011.

[11] W.B. Langdon and R. Poli. *Foundations of genetic programming*. Springer-Verlag New York Inc, 2002.

[12] R. Riolo and B. Worzel. *Genetic programming theory and practice*, volume 6. Springer, 2003.

[13] R. Poli and W.B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[14] R. Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2):123–163, 2001.

[15] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic algorithms, noise, and the sizing of populations. *COMPLEX SYSTEMS*, 6:333–362, 1991.

[16] H. A. Abbass, Xuan Hoai, and R. I. McKay. Anttag: a new method to compose computer programs using colonies of ants. In *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, CEC '02, pages 1654–1659, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Birkan Can and Cathal Heavey. A comparison of genetic programming and artificial neural networks in metamodeling of discrete-event simulation models. *Comput. Oper. Res.*, 39(2):424–436, February 2012.

[18] Salem Fawaz Adra, Ian Griffin, and Peter J. Fleming. An informed convergence accelerator for evolutionary multiobjective optimiser. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 734–740, New York, NY, USA, 2007. ACM.

[19] W. Langdon and W. Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. *Genetic Programming*, pages 73–85, 2008.

[20] S. Harding and W. Banzhaf. Fast genetic programming on gpus. *Genetic Programming*, pages 90–101, 2007.

[21] Julian Francis Miller and Simon L. Harding. Cartesian genetic programming. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, GECCO '08, pages 2701–2726, New York, NY, USA, 2008. ACM.

[22] S.L. Harding and W. Banzhaf. Distributed genetic programming on gpus using cuda. In *Workshop on Parallel Architectures and Bioinspired Algorithms, Raleigh, USA*, 2009.

[23] Zdenek Vasicek and Lukas Sekanina. Hardware accelerators for cartesian genetic programming. In *Proceedings of the 11th European conference on Genetic programming*, EuroGP'08, pages 230–241, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] W. Weimer, T.V. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[25] K. Sastry and D.E. Goldberg. Probabilistic model building and competent genetic programming. *GENETIC PROGRAMMING SERIES*, 6:205–220, 2003.

[26] J.A. Walker, J.F. Miller, and R. Cavill. A multi-chromosome approach to standard and embedded cartesian genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 903–910. ACM, 2006.

[27] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[28] G. Wilson and W. Banzhaf. Deployment of parallel linear genetic programming using gpus on pc and video game console platforms. *Genetic Programming and Evolvable Machines*, 11(2):147–184, 2010.

[29] Garnett Wilson and Wolfgang Banzhaf. A comparison of cartesian genetic programming and linear genetic programming. In *Proceedings of the 11th European conference on Genetic programming*, EuroGP'08, pages 182–193, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[31] Sara Silva and Ernesto Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10:141–179, 2009. 10.1007/s10710-008-9075-9.

[32] W.B. Langdon. Quadratic bloat in genetic programming. In *Proceedings of the Genetic and evolutionary Computation Conference (GECCO-2000)*, pages 451–458, 2000.

[33] Khaled Badran and Peter Rockett. Multi-class pattern classification using single, multi-dimensional feature-space feature extraction evolved by multi-objective genetic programming and its application to network intrusion detection. *Genetic Programming and Evolvable Machines*, 13:33–63, 2012. 10.1007/s10710-011-9143-4.

[34] G. Michael Youngblood, Billy Nolen, Michael Ross, and Lawrence B. Holder. Building test beds for ai with the q3 mod base. In *AIIDE*, pages 153–154, 2006.

[35] Felix-Antoine Fortin, Francois-Michel De Rainville, Marc-Andre Gardner, Marc Parizeau, and Christian Gagne. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 2171–2175(13), jul 2012.

[36] Pygene github repository. http://github.com/blaa/PyGene, Website last visited on August 29th 2012.

[37] Official pyevolve website. http://pyevolve.sourceforge.net, Website last visited on August 29th 2012.

[38] Official pygame website. http://pygame.org, Website last visited on August 29th 2012.

[39] Official pyglet website. http://pyglet.org, Website last visited on August 29th 2012.

[40] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 15–22, New York, NY, USA, 1994. ACM.