Universiteit Utrecht

Faculty of Science
Department of Information and Computing Sciences
MSc in Computing Science

Master Thesis

Automated GUI Testing for games using pseudo-DSL

Andreas Nikas
Student ID: 3816834
a.nikas@students.uu.nl

Universiteit Utrecht

Thesis Supervisors:
Wishnu Prasetya
s.w.b.prasetya@uu.nl

Jurriaan Hage
j.hage@uu.nl

February 2014

ABSTRACT

In the last few years, game development has been going through rapid improvements in terms of graphics quality and game play and, thus, the need for better interaction between gamers and games (artificial intelligence) has greatly increased. As games become more complex, however, so do game states; and game developers now have to work on game engines that are able to provide users with more flexible scenarios. As a result, programming a game to react in unpredictable ways significantly raises the possibilities of facing new challenges, as there is always more space for errors and bugs that the human eye is now unable to perceive, let alone reproduce for testing purposes, within a reasonable period of time.

I firmly believe that evolutionary computing could prove to be a valuable asset towards this direction and play an interesting role in game development and testing, by helping game designers, programmers and testers to better understand the behavior of a game. The advantages of evolutionary programming methods could fulfill the need for high testing standards, since they can produce almost human and less predictable behaviors.

In my thesis, I will research evolutionary computing techniques and examine how they perform in player behavior simulation, as well as develop the corresponding algorithms to given testing purposes. More specifically, this thesis presents a prototype of a new Domain Specific Language to describe a testing purpose and how it could be decomposed into a plan. Then, this needs to be converted into fitness functions, along with an efficient algorithm to solve these functions to produce test-cases satisfying the purpose; which might be difficult for a human tester to solve manually. For my evaluation part, I will develop a tool/library using an emerging scripting language, Lua. My main goal in that part is to prove the scalability of my algorithm and expose wrong behavior, bugs and errors in real game software.

The field of evolutionary game testing is still unexplored and it seems that it can potentially offer a lot to the computing science community. My research could help in reducing time consumption in game testing. It could also improve the Quality Assurance procedure and provide game designers with new ways of developing better AI systems.

PREFACE

First and foremost, this thesis is dedicated to my brother Aleksandros! Without him, I would not have the opportunity to be here!

I would like to sincerely thank my supervisor, Dr Wishnu Prasetya, for his guidance and support throughout this study. My thanks also go to my second supervisor, Dr Juriaan Hage for his feedback and evaluation of my thesis.

Many many thanks go to my wonderful family for their love and support throughout my life. Thank you for believing in me!

Special thanks go to all of my friends who supported me and never let me down, thank you "Αλμοίρα", thank you ncs, thank you Chris!

I am grateful to Maria, Xaroula and Stefania for their support, their endless love and as well as, their delicious food ;)

Andreas Nikas

CONTENTS

1

INTRODUCTION

1.1 CONTEXT

During the last decade, there have been major and rapid improvements in the quality and performance of video graphics cards as well as a fast growth in the game industry: new start-up game development companies enter the marketplace every day -and so do game titles, while the more traditional ones have had to swiftly change and adapt in order to survive the competition.Computer and console video games continuously change the game-play by incorporating more and more features every year. These changes have not only been driven by but also grown higher expectations, in terms of graphics quality, game play, interactivity and AI.

As a result, game design and development of modern games in 3D environment become all the more complex, as game companies strive to achieve the most realistic impression of real-world graphics and conditions, multiplying thus the required parameters and widening the grounds for bugs and errors. Therefore, quality assurance in terms of game testing -which has always been a significant factor to the success of a game title- has become of vital importance.

However, game testing has always posed a great challenge to human testers, locating a bug in a three-dimensional open-world game, with multiple game scenarios and movement freedom, is now more demanding than ever, not to mention isolating the bug and reproducing it in order to determine and fix the bad code segments.

Quality Assurance contributes to the success of a game in the market and it is considered as an important asset of its plain development. Furthermore, many game organizations are willing to invest in hiring skilled engineers and analysts for QA roles and there is a significant percentage increase on QA in the total IT budget of a project (more than 6% than last year)[1].

It is remarkable that a great amount of gaming companies/studios follow different testing techniques in order to assure the quality of their software. One of the most well-known game testing techniques involves the use of Capture/Replay tools (also known as Capture/Playback and Record/Playback) in combination with regression methods(see Chapter 3). An interesting feature of the specific technique is that it uses a scripting language to program and repeat specific actions in loops for various test inputs, enabling

the selection of different routes throughout the process, according to the game results each time. However, aside from other drawbacks of this technique such as the need for system stability and the lack of maintainability, Capture/Replay tools also require manual capturing. Hence, human testing with use of Capture/Replay tools is an ever time-consuming process that requires a high capacity of data to be stored and a large budget to begin with. A particularly noteworthy fact is that more than 50% of the development time is consumed by testing, in order to improve the quality of the software [1].

Currently, there is no significant tool to support an automated testing environment in the level of a system's Graphical User Interface and solve the main game testing difficulties, such as system safety, robustness and usability, even if the companies are trying more and more to use cost-efficient automated tests.

1.2 PROBLEM DESCRIPTION

A fair part of the success pie of software and more specifically of a game belongs to the complete emendation of the technical difficulties. Nowadays, the use of testing techniques and tools becomes more and more necessary.

However, the tools which are used have quite a lot of disadvantages. Capture & Replay, one of the most commonly used techniques, can only be used when the software application is working properly, does not support automatic generation of test cases and requires experienced human resources. Accordingly and in terms of usability, data-driven, keyword-driven and UI-Map-Driven techniques need great experience in scripting languages, and use a lot of files for each test case. The above techniques will be explained in the next session.

As a matter of fact, lack of automation can be observed in each technique. Without taking advantage of that, testing requires not only a significant amount of time and storage but also manual labor and extra costs. An automated technique could provide convenient and faster results in a more timely and cost-effective way.

According to the World Quality Report of 2014 [1], only 13% of the game companies are using purchased automation tools to generate new test data, and another 13% are using custom made automation tools for the same reason.

1.3 OBJECTIVE

My main goal is to research and develop a prototype of a new Domain Specific Language to model the game/'s functionality in order to perform more accurate and less time consuming automated tests. According to [2], a domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on -and usually restricted to- a particular problem domain.

To achieve such automation, I will research and optimize certain evolutionary computing techniques.

1.4 RESEARCH QUESTIONS

RQ:	*How can game related testing goals be expressed in game terms?*
RQ:	*In what terms could a Domain Specific Language improve the automation testing experience?*

Some interesting sub-questions arise out of the first research question:

SQ1:	*Is it possible to break down a game testing goal into sub-goals?*
SQ2:	*Many games involve randomness, concurrency, or multi players, contributing to a high degree of non-determinism. How can a testing goal be solved under such conditions?*
SQ3:	*Are there other kinds of DSLs that can be used for goal-oriented game testing?*
SQ4:	*Is it possible to have a DSL for that purpose, and could it be generic for all types of games?*

1.5 RESEARCH APPROACH

Before proceeding to the development and evaluation of our language and in order to establish the necessity of our purpose, we commit to an in-depth literature review on DSLs that have been developed and used in goal-oriented game testing in the past, therefore essentially answering the third sub question. After going through the literature, the very next step is to research and develop a Domain Specific Language which can describe generic entities of a game. That language should be able to express the initial state, the goals and sub-goals of the game, accompanied by a set of operators and a set

of objects to operate on [3].

However, instead of developing a language from scratch with its own parser and development tools, syntax and documentation, we decided instead to use features of existing frameworks and languages and only add problem-specific keywords, thus creating a pseudo-DSL. A pseudo-DSL is a common DSL which relies on an existing programming language. Consequently, there is no need of writing or maintaining a parser for the specified domain language. The use of pseudo-DSL will save us time from the research, development and debugging of a completely new DSL.

Taking it to the next step, some experiments are carried out in order to prove the usability and scalability of our pseudo-DSL, certain game testing scenarios have to be defined and tested with the proposed language. The scenarios at play will be specific mazes in a 3D environment, which will be generated using a common technique for solving and generating perfect mazes, namely a recursive backtracker algorithm (see Chapter 2.4). The test cases will consist of four levels of difficulty. The following three custom made algorithms, which are the implementations of some keywords of our pseudo-DSL, will be used to solve these mazes:

- *a random walk algorithm*;
- *an Ai-heuristic algorithm, and*;
- *an evolutionary algorithm*;

Eventually, after assessing how all algorithms performed for the given test cases and using the results of the testing procedure, we will be able to evaluate the proposed language entirely as well as conclude whether our game testing goals can be expressed into game terms.

1.6 RESEARCH CONTRIBUTION

In this thesis, a 3D game testing framework is presented, providing fully functional but novel APIs. The framework integrates with the embedded programming language Lua, taking advantage of its syntactic sugar and combining it with our pseudo-DSL. Such a framework, written in Unity3D and Lua, could be used for several purposes. Taking advantage of our pseudo-DSL, we could not only perform model-based testing in a fully-automated way, but also simulate game events through our provided in-game console or via Lua scripts.

Additionally, we provide 3 "monkey" bug tracking algorithms which we used for our experiments and could be found handy for future use, as long as they are fully extensible

and customizable.

Furthermore, we set the foundations for a fully extensible pseudo-DSL for one of the most known 3D game engines in the Industry of game development.

2

TOWARDS GAME TESTING

2.1 CONTEXT

In this Chapter, we introduce basic concepts and background which are relevant for this thesis. In Section 2.2 we mention several types and levels of software testing people typically distinguish in practice.. As mentioned before (see Chapter 1), our research is based on Domain Specific Languages for games, and is introduced in Section 2.3. Later on, in Section 2.4 we discuss a variety of algorithms to generate perfect mazes; these are later needed for the Experiment in Chapter 7; and Section 2.5 discusses evolutionary algorithms, which we later use to implement some part of our DSL (Chapter 4).

2.2 TESTING LEVELS

Before explaining the testing levels, it is noteworthy to mention the two basic classes of software testing, namely black box testing and white box testing [4].

Black box testing (also called functional testing) is the testing process that ignores the internal mechanism of a system or component and focuses solely on the output generated in response to selected inputs and execution conditions.

White box testing (also known as structural testing or glass box testing) is testing by taking into account the internal mechanism of a system or component.

Each type of testing can be derived from requirements and specifications that define the correct behavior, in order to identify the incorrect behavior of the current test. Testing usually ensures the detection of the faults remaining from earlier stages of development, in addition to the faults introduced during coding period. Therefore, different levels of testing are used in the testing process and each level of testing aims to test different units of the system.

According to Laurie Williams [5], there are six levels of software testing: Unit testing, Integration testing, System or Functional testing, Acceptance testing, Regression testing and Beta testing. Though, as stated in Software Testing Fundamentals [**?**], Regression Testing can be considered as a testing technique, rather than level, as long as it can be performed at any of the first four levels.

Unit Testing is the testing of individual hardware or software units or groups of related units [4]. The main purpose of this testing level is to validate that each unit of the software application performs as designed. Units can be considered as the smallest testing part of software and they usually have one single output. It is performed by using White Box Testing techniques and usually carried out by the software developers themselves. This is the first level of testing and ensures the reliability of the code. But, even if the units work individually, that does not mean that all of them combined necessarily work as intended. Integration testing then follows to test the integrated code.

Integration Testing is the testing process in which software components, hardware components or both are combined and tested to evaluate the interaction between them [4]. It can be used along with white box techniques and black box techniques and its main purpose is to expose potential faults in the interaction between combined units. Integration Testing can be performed only by software developers or experienced independent testers.

System Testing is testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements [4]. It is usually carried out in the black box way (the whole internals of a system is too complex to be handled ala white box) and by an external testing team. Its main purpose is to examine the high-level design as well as the customer requirements, in order to ensure that the functionality responds to what the system is meant to do. This level of testing typically also include the testing of non-functional aspects of the whole system, such as its performance and security. Thus, it includes, for example, the following additional types of testing:

- *Stress Testing, which is conducted to evaluate a system or component at or beyond the limits of its specified requirements [4].*
- *Performance Testing, which is conducted to evaluate the compliance of a system or component with specified performance requirements [4].*
- *Usability Testing, which is conducted to evaluate the extent to which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [4].*

Security Testing is a process to determine that an information system protects data and maintains functionality as intended. The six basic security concepts that need to be covered by security testing are:confidentiality, integrity, authentication, availability, authorization and non-repudiation.

System Testing is where most resources are allocated to and it is too timely and expensive to be used in order to locate lower-level faults. Our research will be performed and evaluated on the System Testing level.

Acceptance Testing refers to the formal testing process used to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system [4]. It is performed immediately after System Testing and before making the software available for the end user, using Black Box techniques. These types of tests are pre-specified by the customer within a realistic environment and this level is usually merged with System Testing.

Regression Testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [4]. As mentioned above, regression testing can be performed on each pre-defined level (more usually in the system testing level) whenever the software changes, using both black and white box testing. The purpose of running the regression test case is to make a "spot check" in order to examine whether the new code works properly and has not damaged any previously-working functionality by propagating unintended side effects [5]. Due to the importance of regression testing, companies invest a respective amount of time and money to adopt automated regression tools.

Beta Testing is testing conducted to determine unexpected system faults and is directly connected with acceptance and system testing. Beta testing belongs to the black box testing class and it is performed by potential users or beta testers. Using the specific testing type, companies could identify unexpected errors produced by the users in a variety of environments without any costs, as long as the testers are also users. On the other hand, the lack of systematic testing, the low quality error reports and the time cost to examine the error reports are significant disadvantages.

2.3 AUTOMATED SOFTWARE TESTING TECHNIQUES

As already mentioned, testing is an expensive, time-consuming and intensive process which has to be repeated for every single modification. Automation in testing could give a vital boost in development. The following section describes the most commonly used automated testing approaches.

2.3.1 *Capture and Replay*

Capture and Replay Testing has been one of the most used approaches until now, since almost every automated tool implements that method. All tests are performed manually, by recording all inputs and outputs of the procedure. Immediately after the capturing process, the same sequence of actions automatically replays itself while it logs the new results. After finishing with the replay process, the actual responses to the captured

results are compared and the differences are reported to the tester as errors.
The technique has several advantages over other methods [21]:

- *It requires the least training and setup time, given that it does not require programming skills.*
- *The development of the tests can be defined by the tester on the fly.*
- *Logging the whole procedure of the test cases provides an excellent trail of the steps in order to recreate the error.*

Unfortunately, there are several disadvantages which leads to the development of new methods such as script-based testing:

- *As mentioned before, all tests must be performed manually in order to be captured, resulting in higher time costs; this is even worse when a test needs to be adjusted after a modification to the code.*
- *In order to perform that method, the software must be stable enough for the manual testing / capturing part.*
- *The lack of maintainability of the test scripts is an important factor. Every update in the code that changes the interface may force us to recapture a whole set of tests, thus incapacitating the automation.*
- *The testing scripts are short lived, which directly affects the maintainability of the project.*

2.3.2 *Script Based Testing*

The Script Based Testing approach uses test scripts to automate the execution of the tests. A test script is an executable script which runs one or more test cases and is written in a programming or scripting language such as Lua, Perl, PHP, JavaScirpt, Groovy; Lua will also be used for the implementation of the current thesis. The test scripts are responsible for the initialization of the SUT (System Under Test) and its calibration in the required context. After that, it defines the test input values, passing them to the SUT, and then compare its output to pre-determined expected results. Test scripts must control and observe the SUT using certain APIs [22]. This implies that testability criteria must be defined, that guarantees that such APIs exist.

Script based testing can be combined well with regression testing with the regression testing technique as it solves the execution problem by automating it. That way, the tester could easily run all the test scripts for free.

But, what happens when parameters that are used to initialize the SUT, are changed in the API? Unfortunately, each time some requirements change or some implementation details change, the test scripts must also evolve. This important factor increases the test maintenance problem which results in more costs. Furthermore, test scripts are quite complex which requires experienced programmers in order to write and maintain them.

The next two approaches, namely keyword-driven and data-driven testing, solve the maintainability problem of low level test scripts by raising the abstraction level of the test cases.

2.3.3 *Keyword Driven Testing*

Keyword Driven (or Table Driven) Testing uses action keywords or phrases to express specific fragment of the execution of a test script. The combination of those keywords is translated to executable code which follows the same steps such as the script based testing technique, aiming that way in a higher abstraction level. The implementation of these keywords requires basic programming skills but the design of the test cases can be written by non programmers. The higher level of abstraction does not only offer less production costs, but also offers easy maintenance as the test cases can be used in an updated or modified SUT.

However, the test data are designed manually, and requirements have to be in accordance with the test cases resulting in the necessity of manual tracking, which is still a costly process.

2.3.4 *Data Driven Testing*

The Data Driven Testing technique procedure is quite similar to the previous techniques and can be easily compared with the capture and replay. The only difference is that, in the first one, the inputs and outputs are fixed instead of variables, as opposed to the second one.

Some extra advantages of this technique are:

- *The design of the tests can be accomplished easily. There is no need to have a stable working SUT as the test cases can be developed beforehand. Only the execution of the test requires a running SUT.*

- *The creation of the test cases -inputs and outputs- can be accomplished by anyone and does not require programming skills. All the data can be stored using a simple text editor.*

However, execution of the test cases requires not only experience and programming skills, but also data management and high storage capacity.

2.3.5 *Model Based Testing*

Model Based Testing is one of the most widely used automated techniques that includes the answer for each problem of the previous approaches. It provides automation, not only during the execution of the test but also during the creation of the test cases. That results in the reduction of the maintenance costs. Furthermore, the traceability from requirements to test cases is generated automatically.

The automation of the test cases is an important factor that reduces the production costs. Instead of writing multiple test cases manually, the specific technique generates a set of test cases with regard to a predefined abstract model of the SUT.

According to [22], a model based process can be divided into the following main steps:

- *Model the SUT and/or its environment*
- *Generate abstract tests from the model*
- *Execute the tests on the SUT and assign verdicts*
- *Analyze the test results*

The first step is of vital importance, since it defines the accuracy and stability of the test cases. Modeling the SUT is a manual procedure and the main purpose is to define an abstract model of the system that needs testing. In our case, we will try to define a Domain Specific Language which describes a specific type of game and hopefully a game in general.

Afterwards, a model based related tool automatically generates test cases using the defined abstract model. In that case, the test selection criteria must be carefully considered, so that the number of test cases is significantly reduced to a finite amount.

The third step of the model based process will transform the abstract test cases to executable scripts. For that purpose, the need for a transformation tool is required as it will manage to fill the gap between the abstract test cases and the concrete SUT. This step can be achieved by linking the abstract model terms with low level SUT details such as functionality, movements and interactions. An important advantage of that step is

that the abstract tests could use a different programming language from the one used for the Environment.

The last two steps are a typical part of the testing procedure. First, all the test cases are executed on the SUT while they are logging all the output responses of the software. Afterwards, all the test results are analyzed and a readable report is then produced for the tester. It is important to mention that when a fault is identified, it could be a real fault in the SUT, but it could also be falsely reproduced if the model turns out to be incomplete. A model is complete if every execution of the SUT that is correct, is also verified as correct by the model.

2.4 EVOLUTIONARY ALGORITHMS

For the implementation part of my thesis, I will use evolutionary algorithms in order to generate test cases [13]. An evolutionary algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions (see also loss function). Evolution of the population then takes place after the repeated application of the above operators.

A genetic algorithm takes a random population of solutions as an input and, through a recombination process as well as mutation operations, it gradually evolves the population towards an optimal solution. Apparently, an optimal solution is not always guaranteed and that depends on how well defined the fitness function is. The fitness function evaluates the populations and decides which of them fit as parents of the next generation of solutions.

The resulting solutions with the highest fitness values form the new population and the cycle is repeated, in an effort to approach the optimal solution.

Evolutionary algorithms have been the most popular search-based algorithms for generating test cases and [14] supports exactly that, with a benchmark between 5 techniques for test case generation using a genetic algorithm. The genetic algorithm outperforms the other algorithms in most scenarios and distributions; it is the best choice when there is simply no algorithmic way known to solve the problem (e.g. it is a black box) or the algorithmic way to find a solution of the problem is too costly.

2.5 PERFECT MAZES

With regard to the graphical interface of the game, “perfect” Mazes are used. Perfect mazes are mazes with only one single unique solution that does not involve retracing steps [11]. This means that the maze has no inaccessible sections, no circular paths and no open areas. In terms of Computer Science, such a Maze can be described as a spanning tree over the set of cells [12].

Below, there is a representative description of the most known maze creation algorithms that were researched, in order to pick the most appropriate algorithm for our cause:

- *Recursive Backtracker*:This is one of the most known algorithms in its field which is also directly related to its solving method. The main characteristic of the algorithm is that, each time it moves to a new cell, it pushes the previous cell on the stack. If there are no unvisited cells next to the current position, it pops the stack to the previous position. The Maze is done when the stack is empty. The Recursive Backtracker is one of the fastest maze generation algorithms in the family of perfect Mazes, even though it requires a stack up to the size of the Maze, while keeping the “Dead End” rate really low.
- *Prim’s algorithm:*During the implementation of the Maze, it gives each cell one of the following types: The first type, “In”, means that the cell is already carved and it is part of the Maze; the “Frontier” refers to when the cell is part of the Maze but not carved; and the “Out” type when the cell is not part of the Maze. The algorithm starts with an “In” cell and baptizes the surrounding cells as “Frontier”. Randomly, it selects a neighboring "frontier" which carves into it and updates the neighboring “out” cells to “frontier”. The algorithm stops when all of the “out” cells are gone. It runs really fast and the solution of the Maze is usually pretty straight forward.
- *Kruskal’s algorithm*:It is actually an algorithm used for creating minimum spanning trees from connected weighted graphs. The algorithm requires a storage of proportional size with the size of the Maze, as well as a labeling procedure that enumerates edges and walls between cells randomly. Kruskal's algorithm, then, reviews each edge to detect if the cells on either side have different IDs, deletes the wall in between and labels the cells all over so that they share the same ID. If the cells on either side of a wall already share the same ID, the algorithm assumes there must already be an existing path between the cells at play, and the wall is left intact, in order to avoid an unwanted loop. Furthermore, it runs quite fast providing low solution rate.
- *Aldous-Broder algorithm*:The algorithm does not require extra storage or to keep track of the stack. This algorithm starts from a random cell and moves to a random adjacent cell. If an un-curved cell is entered, it carves it continuing the previous

cell. The Maze is ready when all cells have been carved. The interesting thing of the specific algorithm is that it will create all possible Mazes of a given size with equal probability. However, it is quite slow and the termination of the algorithm is not guaranteed.

- *Wilson/'s algorithm*:This is an improved version of the Aldous-Broder algorithm, the main difference being that it runs much faster. It also only requires storage up to the size of the Maze. It first adds the Maze's first cell randomly, then starts from another random cell that is not part of the Maze and does random walk till it finds a cell that belongs to the Maze. When the already created part of the Maze is hit, it goes back to the random cell that was picked at the beginning and it carves the path it had taken, adding the cells from the path to the Maze.
- *Hunt and kill algorithm*:It is similar to the recursive backtracker algorithm but it does not require extra storage or stack. Therefore, it could implement largest Mazes. The "Hunt and Kill" algorithm starts exactly as the Recursive Backtracker and, when a collision is made with no escape options, it activates the "hunting" mode in which it performs and overall scan of the Maze until a cell not yet belonging to the maze is found next to an already carved cell; the cell is added to the maze and constitutes the starting point of the carving procedure yet again. The Maze is completed when all the cells are scanned by the "hunt" mode at least once. It usually keeps low "dead end" and "solution" rate.
- *Growing tree algorithm*:This algorithm is fully customizable and only requires storage up to the size of the Maze. It starts from a random point in the grid and, each time it carves a cell, it adds it to a list. Afterwards, it picks a cell from the list and carves a path into an uncarved cell next to it, adding the latter to the list. Then the listed cell gets erased from the list and, eventually, the algorithm is done when the list is empty. The interesting part of this algorithm is that customizing the selection of the uncarved cells can simulate other algorithms such as the Recursive Backtracker or Prim/'s.
- *Eller's algorithm*:This algorithm is special because it's not only faster than all the other ones that lack obvious biases or flaws, but its creation is also the most memory-efficient [12], since it only reserves in the memory one generated row at a time. There is a set for every cell in a row, and two cells belong to the same set only if there is a path linking them. The checking process only works for the part of the Maze that has already been carved, thus preventing isolations and loops.
- *Recursive division*:It is yet another algorithm similar to the "Recursive Backtracker" algorithm, with the only difference being that it focuses on walls instead of paths. It starts randomly with either a horizontal or vertical wall, crossing the available area in a random row or column. Afterwards, it repeats the process on the pair of the divided subareas recursively. The "Dead End" ratio is quite high, but it is relatively fast to be created.

- *Binary tree algorithm*:The "binary tree" algorithm is one of the most biased ones, but is also one of the most trivial to implement. Furthermore, it is the simplest to solve and the fastest algorithm to form a Maze. As the name suggests, if starting from upper left side, It could be considered as a binary tree, with its root on the upper left and using only one rule which dictates that each node has one parent -which can be either the upper left cell or the cell above, but never both of them. Reversing a binary tree Maze upside down, treating the passages as walls and the walls as passages, you get another binary tree.
- *Sidewinder algorithm*: This algorithm also generates the Maze by one row at a time. It starts by randomly deciding whether to carve a passage leading right or considering the horizontal passage completed. Then, it randomly picks one cell along this passage, and carves a passage leading upwards. In comparison to the binary tree algorithm which goes up from the leftmost cell of an horizontal passage, the sidewinder algorithm goes up from a random cell. Thus, the solution rate is low and almost the same as the binary tree algorithm.

Algorithm	Dead End %	Type	Focus	Bias Free	Memory	Time	Solution
Recursive Backtracker	10	Tree	Passage	Yes	N^2	24	19.0
Hunt and Kill	11	Tree	Passage	No	0	55	9.5
Recursive Division	23	Tree	Wall	Yes	N	8	7.2
Binary Tree	25	Set	Either	No	$0*$	7	2.0
Sidewinder	27	Set	Either	No	$0*$	8	2.6
Eller's Algorithm	28	Set	Either	No	$N*$	10	4.2
Wilson/'s Algorithm	29	Tree	Either	Yes	N^2	51	4.5
Aldous-Broder Algorithm	29	Tree	Either	Yes	0	222	4.5
Kruskal's Algorithm	30	Set	Either	Yes	N^2	32	4.1
Prim's Algorithm	36	Tree	Either	Yes	N^2	21	2.3
Growing Tree	49	Tree	Either	Yes	N^2	43	11.0

Table 1: the main characteristics of the perfect Maze creation algorithms

.

The table above shows the main characteristics of the perfect Maze creation algorithms [12]. These numbers are taken from a $100x100$ Maze using the software Daedalus.

The column Dead End contains the approximate percentage of cells that are dead ends in a Maze created with this algorithm. The table is sorted by that column. The recursive backtracker algorithm supposedly could have less than 1% dead ends, having enough run factor and the highest possible dead end could go up to 66%.

Furthermore, there are two types of perfect Maze creation algorithms. Tree bases algorithms grow the Maze like a tree, having always a valid perfect Maze on every step of the algorithm. On the other hand, a set based algorithm builds where it pleases, keeping track of the parts that are connected with each other, in order to form a valid Maze at the end of the procedure.

The 'Recursive Backtracker' algorithm and the 'Hunt and Kill' algorithm cannot form a Maze by adjusting walls on their passage but only through curving their path, contrary to the 'Recursive Division' algorithm where it can only be done using a wall adder due to its bisection behavior. The rest of the algorithms could work both ways adding walls or curving their path.

The Bias Free column means that the algorithm treats all directions and sides of the Maze equally, where analysis of the Maze afterward cannot reveal any bias.

An important factor in a perfect Maze creation algorithm is how much extra memory or stack is required to form it. Some of the algorithms do not need any memory storage, while others need a single row (N) or proportionally to the number of cells (N^2).

The Time column shows the time it takes to create a Maze using this algorithm and the numbers are only relative to each other and depend on the size of the Maze. Some of the algorithms that either use walls or curving to implement the Maze take much longer when they add walls. Last but not least, the Solution column is the percentage of cells in the Maze that the solution path passes through with the start and the end in opposite corners.

3

DSL IN GAMES

3.1 CONTEXT

In this chapter, I will try to give a short overview of the work that is associated with the area of domain specific languages used in game testing automation with a model-driven approach. Before going through the related literature, it should be pointed out that there is not really anything related to the field of game testing, as far as domain specific languages are concerned, but only in that of software testing.

To begin with, the most known domain specific languages that are used to express game terms are listed below. Even though these Domain Specific Languages were primarily developed in order to express a game or even construct the functionality of one and not actually test it, they inspired our attempts nevertheless. It should also be made clear that these languages are designed to help develop 2D games and not 3D ones. In the second section, I present the most used -state of the art- approaches in automated software testing.

3.2 DSL

This thesis is directly related with the research and design of abstract models that describe a game, or a test case of a game. A Domain Specific Language (DSL) can be used to describe models. But first, what is a DSL, and what are the advantages and disadvantages of using it?
According to [6] and [7], a DSL is a programming language specialized to a particular application domain that offers, through appropriate notations and abstractions, expressive power focused on -and usually restricted to- a particular problem domain. A DSL could describe an application domain in a higher level of abstraction which makes it much easier to deal with. As stated in the definition, DSL can be considered as an excellent candidate for our purpose.

One of the most important reasons of using a DSL is the better maintainability that offers, as long as the code is understandable and, more or less, self-documented because of the higher abstraction level. Thus, it gives the opportunity to allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain, making that easier to user with no programming skills to understand, modify, extend and reuse the language.

Furthermore, according to [6], that results in higher quality, productivity, reliability, portability and reusability. Another advantage is that DSLs allow validation at the domain level, as long as the language constructs are safe.

On the other hand, in our case, there is one main disadvantage. Designing and maintaining a DSL is a costly procedure. It takes time not only to design a well defined DSL, but also to train users to use it in a proper way, which could affect the productivity.

As pointed out in [8], using a DSL may lead to a loss in efficiency of the final software, but it may also lead to an increase. When using a DSL, one writes code at a higher abstraction level; translating this code into a general purpose language or some other target language may lead to inefficiencies in the final product. Nevertheless, in our case, we will use a DSL to express a game in order to construct a model which will test it, instead of developing the whole game.

In our case, we took advantage of the programming language Lua. Lua is a powerful, fast, lightweight, embeddable scripting language [9] which was designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. It combines procedural syntax with powerful data description constructs based on associative arrays and extensible semantics.

Lua is a relatively new language which joined the game development from its early stages. We decided to take advantage of Lua by taking into account the following reasons.

First of all, Lua is a proven, robust language and it is considered to be the fastest language in the realm of interpreted scripting languages [9]. But the most important reason is that Lua can be embedded into almost every type of applications. Furthermore, Lua has a simple and well documented API, providing examples and special programming cases.

Last but not least, Lua is malleable. It has just enough syntactic sugar and meta-mechanisms to be easily repurposed for domain specific languages [10].

There is a great variety of DSLs known as domain-specific entertainment languages. Based on [15], these are a group of DSLs that are used to describe computer games or environments, or even potentially used for other digital entertainment such as video or music. Some of the more well known examples of these are: Extensible Graphical Game Generator, Zillions of Games, ViGL, Py-VGDL, Game Description Language (GDL), GDL-2 and World Description Language, UnrealScript, GameXML, and Xconq. The following subsections explain some of them, which are most used in the game industry.

None of these are used to test games actually, but they at least prove that terms and concepts in games can be abstracted with a DSL. For each of these DSLs of the following sub-chapters, there is a code sample of the well-known game “tic-tac-toe” at the chapter appendix, taken from [8].

3.2.1 *GDL & GDL-2*

To begin with, it is worth mentioning the term of general game playing was first introduced by the Stanford Logic Group of Stanford University and is the reason why one of the most known description languages for games was developed, namely Game Description Language (GDL) as well as its extensions, such as GDL-2. "General Game Playing" is the design of artificial intelligence programs in a way that enables them to play more than one game successfully [16]. According to [17], games are defined by sets of rules represented in the Game Description Language.

GDL describes the state of a game world in terms of a set of true facts [18]. It uses logical rules in order to define the set of true facts in the next state, and keeps track of the transition between the states. It consists of an initial state, a goal and a terminal state. GDL and all of its extensions like GDL-2, can be natutally applied to board and logic games with a two dimensional grid system, as these games have a deterministic set of states and and moves that are expressible as a finite state machine.

We tried to express the game logic of a simple 2D shooting game, given that GDL can be applied to a 2D grid. We managed to set the basic view definitions and rules such as the role, initial, goal, terminal and basic legal relations; but the amount of the static relations such as the definition of each integer or the definition of the successor of each integer stopped us from further experimentation.

The following table shows the basic keywords[19] of GDL-2 that we used for our attempt.

3.2.2 *ViGL*

The domain-specific language ViGL stands for Video Game Language. It is mainly used for rapid prototyping on 2-dimensional games. Expressions in ViGL can be used to generate the backbone code of a game.

ViGL is based on XML which makes it easy to understand but it also lacks in control flow features. Thus, at the end, the ViGL designers mixed XML with some embedded code (XML blocks).Each XML block describes a common component of a video game.

role (R)	R is a player
init (F)	F holds in the initial position
true (F)	F holds in the current position
legal (R, M)	R can do move M in the current position
does (R,M)	Player R does move M
next (F)	F holds in the next position
terminal	The current position is terminal
goal (R,N)	R get N points in the current position
sees (R,P)	R perceives P in the next position
random	The random player
distinct	Is used to require that two terms be syntactically different

Table 2: GDL-2 commands

.

The ViGL specification focuses on the graphical needs of the game and the integrated game rules of each game object.

In principle, ViGL only targets 2D games, though according to Jeroen Dobbe [8], its features(see Table 3) can in principle be translated to a 3D setup.

Features	Requirements
Graphics	Loading of images, texts, etc
Sound	Load, Play, Pause and stop music
User Input	Keyboard and mouse input handling
Objects	The objects of the game(types, shapes, sizes, positions)
World	The game world and the rules of the world
Interaction	Interaction between object in the game world.
User control	The way the user controls the objects in the world

Table 3: features that can be translated to three dimensional environment

.

The table above suggests out that a DSL for games should not only express the high level game design definitions but also lower level details of graphics, sound, interactivity etc.

An important aspect of ViGL is that it provides classes and methods for the game objects, the rules, the world, the events and the actions. On the other hand, game play as-

pects such as the player types, world types, scoring system and levels are missing in ViGL.

3.2.3 *Zillions of Games*

Zillions of Games is a commercial game package with a universal game engine for board games and was developed by Jeff Mallet and Mark Lefler in 1998. It was designed to handle mostly abstract strategy board games or puzzles. After parsing the rules of the game in the game engine, the system could provide solutions by automatically playing one or more players with the use of artificial intelligence.

The scripting language of the Zillions of Games uses S-expressions which makes the code more structured and easy to understand. S-expressions, which stand for symbolic expressions, are a notation of nested list data, used for the programming language Lisp. An S-expression is defined as an infinite set of distinguishable atomic symbols or as an expression of the form $(x.y)$ where x and y are S-expressions and the parenthesis represents an ordered pair so that S-expressions are effectively binary trees [20].

A game definition by Zillions of Games can be separated into four basic parts. The first part contains the name of the author, the game description and other metadata of the game. The second part of the structure contains the definitions of the objects of the world, the players and its meta information. The third part contains the board information such as the shape, the grid style and dimensions. The last part of the definition contains the goal of the game, expressed in just a condition where one of the players wins or when draw occurs.

4 METHODOLOGY

4.1 CONTEXT

This chapter presents the methodology chosen to answer the posed research questions (see Section 1.4). Since we have two research questions, the methodology is split into two corresponding parts.

The first part of our approach is the design of a DSL to express testing related concepts for games, and the development of an implementation that facilitates more accurate and less time-consuming automated tests. It would save us a lot of effort if we could work on an existing DSL. Unfortunately, all DSLs used in the Game Industry, which we discovered during our literature study, are used solely for developing games, rather than for testing them. Therefore, we have to design our own.

A DSL is a small, usually declarative language that offers expressive power, focused on a particular problem domain ([23], which is a more precise definition of [2]). In a DSL used for expressing testing goals for games, we will inevitably need to express these goals in terms of the games' concepts and terms. In addition, pseudo-DSL includes commands and queries of the domain which can be syntactically expressed with the help of a programming language. A pseudo-DSL contains pseudo-code which is an informal high-level description of the operating principle of a computer program or other algorithm [24]. It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.

There are however various types of games, such as first and third person action games, role-playing games, strategy and massively multi-player online games, that greatly differs from one another. Despite some common concepts like that of victory and defeat, each type seems to have a whole range of unique concepts, such as game-play, physics, graphics and rules; thus essentially defines its own domain. We will therefore focus on one type, namely 3D First Person Shooter (FPS) games, and our DSL will be one for testing these types of games.

In order to avoid the development of a new DSL from scratch, an effort to extend an already existing DSL such as the GDL-2 language was made but without any success. As already discussed (see Chapter 3.2.1), GDL-2 is not designed to test games but to develop turn-based strategy games in a two dimensional environment. Thus, our next

alternative step was to take advantage of an existing embeddable scripting Language such Lua and try to express game terms through that. According to [25], an embedded style language is a kind of computer language whose commands appear intermixed with those of a base language.

Embedding Lua in the Unity3D Game Engine gave us the opportunity to not only express the testing goals in game terms but also control the whole game through Lua, which resulted in the development of a pseudo-DSL instead of a complete DSL.

Concerning the evaluation of the first part of our approach and the answer of RQ2, we decided to apply experiments by using three different algorithms on four different levels of difficulty via our pseudo-DSL. The algorithms of the experiments will be presented in the following sections of this chapter.

Furthermore, it is wise to present the basic structure of the domain in combination with our implemented pseudo-DSL. Figure 1 shows the architecture of our pseudo-DSL, and how it is embedded in the Unity game engine.

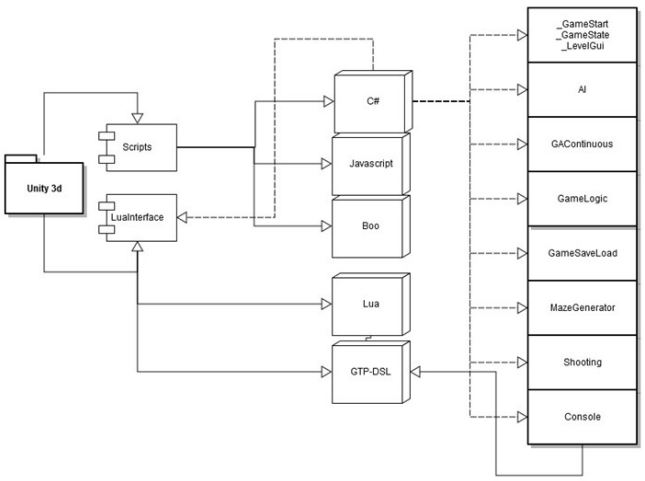

Figure 1: Basic Unity3d structure

In Unity3D, a game is built by providing scripts and scenes. Scripts express various functionalities of the game, e.g. the functionality to shoot, and the functionality to save and load a game in progress. Scripts can be written in C#, JavaScript and Boo. We extend Unity3D with a LuaInterface, which is implemented as a Unity script, written in C#. This interface is responsible for the interpretation of Lua and the communication between the domain and the pseudo-DSL.We also added a so-called "console", which is

a graphical user interface that allows the tester to directly query the game when it is under test. We found this to be a very useful tool. More detailed information will follow in the following sections.

In Section 4.2, I present the reasons why we used the specific game engine and its structure as well as the rules and controls of the game, while, in Section 4.7, the reasons for selecting a more specific domain such as mazes are discussed. Section 4.3 and 4.5 introduce Lua and the console which was used to communicate with the domain. The implementation of our pseudo-DSL, as well as the algorithms that were used in order to test the usability and scalability of our language can be found in sections 4.4 and 4.6 respectively. Last but not least, Section 4.8 presents the problems we faced during the procedure of the research and how we managed to solve them.

4.2 GAME DEVELOPMENT WITH UNITY3D

We decided to develop our domain using the well known game engine, Unity 3D version 4.0.1 for Windows Operating Systems. Unity3D is one of the most widely used game engines, and some of its typical features are the following:

- *Unity3D is free for personal and commercial use and provides a wide range of features.*
- *It provides developers with many examples and features a strong, community-supported API.*
- *It is quite easy to use for 3D and 2D game development purposes.*
- *Developers are free to use any one of a wide range of programming languages, like C#, JavaScript, or Boo.*
- *More importantly, as far as this thesis is concerned, many more languages can be embedded into the engine, like Lua.*
- *Unity also provides reliable efficiency and frame rate, which is really important when testing a game.*

The next figure presents the important components that Unity 3d consists of, accompanied by our domain which will be analyzed later on.

Figure 2 shows the important components of our extended Unity3D, namely: Scripts, Scenes and LuaInterface. The component Scripts contains C# and Lua scripts which are responsible for the front-end and back-end domain functionalities. The most important of them are discussed in the following Sections. The component LuaInterface links and interprets the programming language Lua to the Unity 3d environment. Last but not least, the component Scenes contains the front-end GUI and the back-end Maze Generator

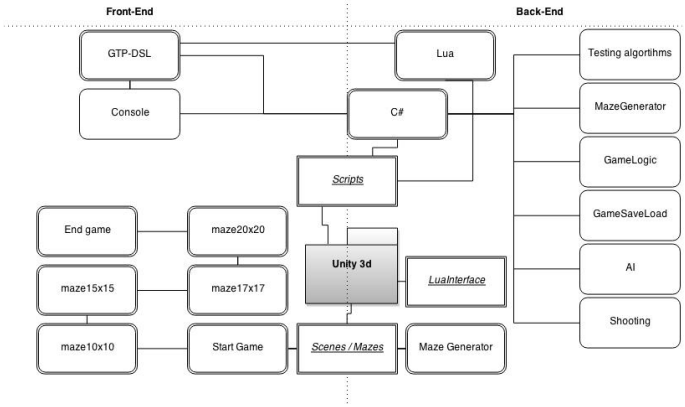

Figure 2: Unity3d structure

scene which will be explained in Chapter 4.7.

In order to limit the scope of my thesis, I created a basic First Person Shooter (FPS) game, the primary goal of which is the detection and neutralization of one or more targets by a player. This way, we could represent a real game, providing basic 3D graphics and controls.

We tried to set as simple game rules as possible, so that we could relate them to an FPS game. In the game scene, there is one player and one or more targets - enemies. The enemy can move randomly across the stage and the player must locate it running through a maze of a certain size. The game goal is to locate and neutralize the target as fast as possible. It should also be noted that there is a time limit for every stage of the game. The game stops when the player has neutralized all targets - goals within the time limit, when the time limit has been reached, or when the player is found outside the stage, which means that a certain bug has been detected.

The player's movement in the game can be controlled in three ways. The first one involves the keyboard: forward ('W'), backwards ('S'), left ('A'), right ('D'), and shooting can be achieved by left-clicking on the mouse.

The other 2 ways can be accessed through a Quake-like console that was built as part of our extension of Unity3D?. They provide the player with information about the game state, as well as complete management of the objects at the current game state. The console commands were also used in the pseudo-DSL. The console can be activated using the (`) key, and its functionality will be thoroughly discussed in the following sections.

4.3 CONSOLE

The console has been developed, for various reasons, from the beginning of the thesis research. It was vital to know that it is feasible to interact with the SUT at any time. That way, the SUT state is fully controllable and provides feedback for bugs and system errors as well as debugging purposes. Furthermore, the most important feature of the console is being able to interact with the gameObjects of the current state. That manipulation could be achieved by using our DSL, executing functions directly via the console or by using Lua scripts which are actually a combination of the syntax of Lua and our DSL.

We did not develop the console from scratch, but instead extend "In-Game Console" extension [26]; this saves much work.

- *The extension is open-source and free to use.*
- *The code is well-structured and clean.*
- *It provides pretty straight forward functionality to register custom commands*
- *It provides a logging system which stores data into log files*

Consequently, the only part that was needed to be extended was the registration of our custom commands which would also interpret Lua syntax commands with our pseudo-DSL.

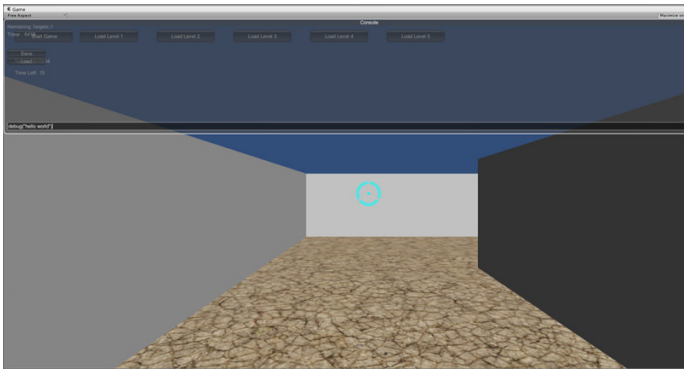

Figure 3: Console activation example - upper semi transparent box

4.4 THE PSEUDO-DSL GTP-DSL

The lack of literature in the field of Domain Specific Languages for Game Testing lead the research to Game Description Languages for logic and board games, which could only express game terms in a 2D environment. However, these are the reasons which motivated me to develop a new DSL and contribute an updated approach of Game

Testing. It worth mentioning, that the GDL-2 language gained our attention because of the way it was defined. It describes the state of a game in terms of a set of true facts by providing keywords to the tester.

Furthermore, by taking advantage of the Lua syntax, we managed to develop the following pseudo-DSL which can provide the tester with full access to the state of the game. The DSL we propose is a set of keywords which could be applied to every type of gameObject in the environment of Unity3D. Keywords can subsequently be composed using Lua syntax.

In the following Figure can be seen an example of that, which was used for creating test cases using the pseudo-DSL in Lua:

```
1  co = coroutine.create(
2      function()
3          goto("Player","Target",3600)
4          coroutine.yield()
5          shootobject("Player","Target",1)
6          coroutine.yield()
7      end
8  )
9  coroutine.resume(co)
10
```

Figure 4: Pseudo-DSL example in Lua

In the script above, we can see a Lua script along with the pseudo-DSL that we have developed. The use of coroutine enables us to execute script files on the fly without having to deal with the known continuation issue that goes with script languages. The *coroutine.yield*() function can be considered as a combinator. As we can see in the example, the gameObject-player will try to locate the gameObject-target within an hour. When either the time is over or the gameObject-target has been located, it will continue to the next function, where it will try to neutralize it.

The keywords/functions which we used to describe the domain can be found in the following table:

Keyword/ Function	I/O types	Location	Description
spawn	GameObject -> Float -> Float -> Float -> String String -> Float -> Float -> Float -> String	Console Lua	Using this particular command, the user can make any given object of the game appears in the map, at the desired location. If this is successful, a success message is returned.
moveto	GameObject -> Float -> Float -> Float -> String String -> Float -> Float -> Float -> String	Console Lua	This command will transfer a game object to the desired location (specified by coordinates) and will return an appropriate success message. This kind of transfer ignores all collisions; so the moved object may stick of pass through other objects within the scene.
fire	Void -> String Void	Console Lua	This command will cast a ray (raycast) towards the center of the player's view. If the raycast collides with an object and this object is stated as a target, then it will destroy it and the total enemies counter will go down by 1. Following that, a success message will be returned in the console.
move	GameObject -> Direction -> Int -> String (Direction = “Left” | “Right” |”Up” |”Down”) String -> String -> Int -> Void	Console Lua	The 'move' command is one of the most useful ones in the game. As input variables, it takes the game object we want moved, the direction toward which we want it moved and the time interval for which we want it moved in space. When the movement is complete, a success message is returned with the new coordinates of the game object, at its new location. Similar to the 'moveto' command, the 'move' command is not subject to collisions.
whereis	GameObject -> String String -> String	Console Lua	Using the whereis command, we can query the position of the object.
canmove	GameObject -> Float ->String String -> Float -> String	Console Lua	canmove is another query command which checks whether it is possible for an object to move to the specific coordinates. More specifically, it applies a raycast for n distance and checks if there is an obstacle in front of it. The parameters that it needs are the gameObject and the number of the steps (width of the raycast).
goto	GameObject -> GameObject -> Float -> String	Console	It takes as parameters two

	String -> String -> Float -> Void	Lua	gameObjects and time in seconds. The first gameObject is the object that we need to move towards the second gameObject for n seconds. Furthermore, this is a kind of an AI-Heuristic algorithm where the first two string parameters are the gameObject-player and the gameObject-target. The last one is the time in seconds. The specific algorithm "knows" the position of the gameObject-target and will always try to move towards it. Furthermore, it keeps in list the last 100 coordinates of its position in the scene, avoiding movement loops.
shootobject	GameObject -> GameObject -> Float -> String String -> String -> Float -> Void	Console Lua	Another shooting command with one more extra functionality. The first gameObject is used to look at the second gameObject in n seconds and shoot afterwards.
debug	A -> String	Lua	The debug command was created so that the user can be informed of the state during an event, as well as the values of any parameter in the console.
loadlvl	Void	Lua	Loadlvl will load the last saved state from file. The save state will contain only the position and rotation of the player.
Evomove	String -> [String] -> Float -> Void	Lua	This custom made evolutionary algorithm takes three parameters: the gameObject-player in string; a list of possible moves such as left, right, up, down; and the time or steps necessary for the algorithm. The use of the list of moves is important, since it defines the events of each population
Randomwalk	String -> -> String -> Float -> Void	Lua	This algorithm requires exactly the same parameters as the "goto" algorithm and, also, every movement in the scene is random.

The first column shows the name of the function, the second defines the type, and the third column indicates the way of executing the specific function. As we can see, most of the functions can be typed and executed from the console which is in the SUT or by a Lua script. It should be noted that the script can be edited on the fly. The last column describes the purpose of each function.

It is important to mention that we took full advantage of the Lua language. This way, we avoided having to define loops, statements and data types, except for a few helper functions addressed to controlling the game. As mentioned before (see Section 4.1), in order to prove the usability and scalability of the proposed pseudo-DSL, the functions "evomove", "goto" and "randomwalk" were used to gather the experimental results.

The console, besides its use that was discussed above, was also used to report errors, warnings and bugs, as well as to keep record of all feedback with regard to executed commands.

4.5 LUA & LUAINTERFACE

Our pseudo-DSL includes only keywords or functions missing the proper syntax of a programming language. Hence, Lua comes to fill that gap and provide full access to its syntactic sugar such as operators, loops, conditions and functions.

Lua is a popular and growing programming language in the game industry. According to [27], Lua is a dynamically typed language that offers support for object-oriented programming, functional programming and data-driven programming; and it works embedded in a host client.
More specifically, Lua provides us with the following elements:

- *Error Handling: all Lua actions start from C code in the host program calling a function right from the Lua library so that, whenever an error occurs, it prints it out.*
- *Coroutines: A coroutine in Lua represents an independent thread of execution.Coroutines played an interesting role in our research, helping us to open threads in the same instance of the system and execute functions simultaneously. The Figure 5 presents the proper syntax of a coroutine* Results are presented at Figure 4
  The function print first calls the coroutine co providing it with the parameters $a = 1$ and $b = 10$. Immediately after, the function foo is called with $a = 2$. First, function foo prints the first row of Figure 4 and then yields the coroutine and returns 4. Once the coroutine is "paused" the print function prints the second row

```lua
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
      local r = foo(a+1)
      local r, s = coroutine.yield(a+b, a-b)
      return b, "end"
end)
print("main", coroutine.resume(co, 1, 10))
```

Figure 5: Lua example

foo	2	-
main	true	4

Table 4: One run results

.

of Figure 4.
If we executed the function print again two more times, the end result is presented at Figure 5:

foo	2	-	-
main	true	4	-
main	true	11	-9
main	true	10	end

Table 5: Three run results

.

- *Standard Lexical Conversions: As the Lua guide implies, Lua is a free-form language which ignores spaces, new lines and comments between lexical elements (tokens). The following table presents the keywords that are reserved by Lua and cannot be used as names and variables:*

and	break	do	else	elseif	end
false	for	function	goto	if	in
local	nil	not	or	repeat	return
then	true	until	while		

Table 6: Standard Lexical Conversions

.

It also provides the standard functionality of the following symbols: +, -, *, /, %, ^, #, ==. =, <=, >=, <, > , =, (,), , , [,], ::, ;, :, ,, . , .., ...

- *Statements: Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignments, control structures, function calls, and variable declarations.*
- *Expressions:*Last but not least, Lua provides us with the following basic expressions
  $exp ::= prefixexp$
  $exp ::= nil|false|true$
  $exp ::= Number$
  $exp ::= String$
  $exp ::= functiondef$
  $exp ::= tableconstructor$
  $exp ::= ...$
  $exp ::= expbinopexp$
  $exp ::= unopexp$
  $prefixexp ::= var|functioncall|(exp)$

In order to integrate Lua with Unity3D, LuaInterface was used. LuaInterface, written by Fabio Mascarenhas from department of Informatics in PUC-Rio, is a library used for integrating the Lua language with the Microsoft *.NET* platform's Common Language Runtime [28]. Additionally, we had to compile 2 DLLs, namely luanet.dll and luainterface.dll. Then, programming in Lua was pretty straightforward.

LuaInterface provides 2 different ways of running Lua code:

- After activating the Lua interpreter via C# as follows:
  $Lualua = newLua();$
  The coder can call or use any Lua function or variable through the Unity3D platform. This way was mostly used for coding scripts for the evowalk algorithm on the fly. Saving files with the scripts and, eventually, automatically keeping those that have detected a correct solution or a certain bug in the game allows us to claim that we have partly solved the regression testing problem. With every update in the code, we can verify the validity, running the specific scripts.
- By executing external Lua scripts.

By developing such scripts, we managed to automate the process of the experiments, putting these scripts in a simple loop.

4.6 TESTING ALGORITHMS

Note that we do not necessarily want to use a smart path finding or maze solver algorithm as a test algorithm. The ultimate goal is to find bugs, which may be lurking in less optimal or even in wrong paths, which human users can do. For this reason, none of the algorithms we implemented are aware of the problem's best solution. Furthermore, the Evowalk algorithm is also used as an automated test generator.

4.6.1 *Random Walk Algorithm*

This algorithm uses a coroutine which applies 3 surrounding sensors to the gameObject-player. The gameObject-player starts by applying a forward movement force on itself. In case of collision with any obstacle but not an obstacle with tag "Target", the algorithm decides randomly whether to turn left or right, by 90 degrees.

In order to detect the surrounding non-target obstacles, the gameObject-player applies one forward raycast and two more with an angle of 45 degrees; one on the left side and one on the right side of the gameObject-player. For the given time, the algorithm will try to achieve its assigned goal.

```
Select randomly between left and right and turn
while there is time left do
    Start moving the gameObject-player forward (forward force)
    if the applied raycast hits an obstacle then
        if the obstacle is the gameObject-target then
            Exit
        end
        else
            Select randomly between left and right and turn
            Select randomly whether to look towards the gameObject-target or not; if
            yes then turn till you look towards the gameObject-target
        end
        if the left side raycast or the right side raycast cannot hit an obstacle then
            Rotate the gameObject-player randomly, left or right by 45 degrees
        end
    end
end
```

Algorithm 1: RANDOM WALK ALGORITHM

4.6.2 *Heuristic AI Algorithm*

This algorithm uses the coroutine "GoToHelper" which also applies 3 surrounding sensors to the gameObject-player and a forward force to itself. While the gameObject-player moves towards the gameObject-target, the algorithm scans for obstacles and already visited paths which are stored in an array, preventing the gameObject-player to repeat the same path.

The raycasts work in the exact same way as they work in the Random Walk Algorithm but the left and right raycasts are applied vertically (90 degrees apart) on the gameObject-player. The specific algorithm holds the knowledge of the position of its goal and will always try to move towards these coordinates.

```
Create an empty set for Visited Coordinates
Randomly select between left and right and turn
while there is time left do
    Start moving the gameObject-player forward (forward force) only if the next step
    is not in the List of visited coordinates
    if the applied raycast hits an obstacle then
        if the obstacle is the gameObject-target then
            Exit
        end
        else
            Select randomly between left and right
            Rotate left or right
            Select randomly whether to look towards the gameObject-target or not
        end
        if the left side raycast or the right side raycast cannot hit an obstacle then
            Rotate the gameObject-player randomly, left or right by 45 degrees
        end
        else
            Give the possibility to turn towards the gameObject-target 33.3%
        end
    end
end
```

Algorithm 2: HEURISTIC-AI ALGORITHM

4.6.3 *Evowalk Algorithm*

The evowalk algorithm uses a genetic algorithm that mimics natural processes in evolution such as mutation and selection. Like all GAs, it begins with a random population of solutions, the chromosomes. In our case, the chromosomes are just a sequence of movements pointing the gameObject-player where to go (left, right, up or down). In order to calculate the cost, we used our goal as a fitness function. The goal in our case is to bring gameObject-player to gameObject-target close enough to shoot it.

For that cause, another helper function,which is called ***calculateCosts***, plays an interesting role. Not only does it calculate the cost of each chromosome but also stores our test models in an Array making the logging and bug tracing part easy. Afterwards, it saves the current state of the game and evaluates the population. For each chromosome in the population, calculateCosts calculates the cost by executing the movement sequence and calculating the difference between the position of the gameObject-player and gameObject-target. At the end, it reloads the initial state to prepare the algorithm for the next chromosome evaluation. The ***calculateCosts*** returns a list of costs to the Evowalk algorithm.

These costs will be ordered by their value and mated using single point crossover and mutation. Mutation and crossover will be applied only to the two chromosomes with the minimum cost which will generate the new population. The new population will be evaluated by the ***calculateCosts*** function again following the same procedure.

The algorithm will finish after a specified amount of iterations, presenting the best solution for the specified goal.

Create random initial population
Call the function *calculateCosts*
Sort the costs and obtain indices
Sort the population according to costs
Store the best (minimum) cost in a list
Store average cost in a list
while $(cost > 0)||(countofiterations <= predefinedmaximumnumberofiterations)$
do
  Weight chromosomes
  Perform mating using single point crossover
  Mutate the population
  Call again the function calculateCosts in order to evaluate the new offspring and mutated chromosomes;
end
Return the best solution

Algorithm 3: EVOWALK ALGORITHM

1.Create an empty table for costs
2.Store the test models and for each model calculate the cost by evaluating the model

Algorithm 4: F0 HELPER

4.7 MAZES: A MORE SPECIFIC DOMAIN

As already mentioned, experiments needed to be run in order to measure the capabilities of our pseudo-DSL, and prove the fact that game testing goals can be expressed in game terms and that such a pseudo-DSL can be applied to different scales of the domain (scalability). Thus, we had to specify a domain which is adjustable to different scales in respect of difficulty and complexity.

We choose to use perfect mazes as the domain. A perfect maze is defined as a maze which has one and only one path from any point in the maze to any other point. This means that the maze has no inaccessible sections, no circular paths and no open areas. Moreover, a perfect maze can scale up in respect to width and height, is a challenging environment; they are a challenging enough environment with non-trivial solutions and, thus, is a perfect host for seeding visual bugs, such as passing through or over a wall. Perfect mazes will be discussed in more details in Chapter 5.

Last but not least, the simple player controls of the game in the specified domain resulted to the forming of the following game testing goal: gameObject-player must locate gameObject-target in N seconds and shoot it.

4.8 PROBLEMS AND SOLUTIONS DURING THE IMPLEMENTATION

During the development procedure of the whole game environment, we came across three basic problems; two of them were unexpected.

4.8.1 *LuaInterface*

In order to integrate Lua in the Unity game engine, it was required that an interface be developed but, as already mentioned and due to limited time, an already built interface was eventually used, which expectedly brought up several issues until it was customized to fit our needs.

4.8.2 *Continuation & coroutines*

During the experiments, many issues were observed, with regard to the player's behavior at the moment it was executing the moves needed to achieve the assigned goal. As a scripting language, Lua will execute each command line without waiting for any confirmation. As a result, any change to the game state (like loading the stage all over) would be executed, ignoring whatever commands had yet to be fully executed; this would lead to unexpected bugs.

In order to solve this particular problem, coroutines provided by Lua were applied. This way, we not only achieved a complete control over the continuation of events but also get to query the state of the game, as well, such as the player's location or the player's distance from the goal, at any time. It should also be noted that the coroutines were partly used as a combinator.

4.8.3 *Time consuming algorithm and sub-goals*

Above the level of 50% difficulty in a maze, it was observed that the time needed to find the solution was quite high, meaning that the evowalk algorithm would by far exceed the given time limits.
However, the above issue constituted the main reason why sub-goals were applied, thus replied to the second research question, as well as some of the sub-research questions. Using sub-goals, we managed to exceedingly reduce the experiment time, making it comparable to the other algorithms.

5 EXPERIMENTS

5.1 CONTEXT

In order to prove our concept of pseudo-DSL, we carried out a series of experiments, the results of which are presented in this chapter. The main question we would like to establish is whether our pseudo-DSL can be applied in a way that enables us to express game testing goals in different scales with success. Besides the usability of the pseudo-DSL, an important aspect that we will examine and prove is its scalability; we will prove that it is suitably efficient and practical when applied to complex conditions such as a large number of goals or bigger environments. Last but not least, the algorithms that are tested will also unfold some capabilities of the pseudo-DSL.

As already mentioned, in order to be able to detect the target and accomplish the goal, the user will have to travel across the whole maze in which each stage takes place. Using the Recursive Backtracker algorithm that was discussed in Chapter 2, four mazes of different difficulty levels were developed.

Furthermore, Section 5.2 presents the reason why we choose the specific algorithm and describes -using pseudo-code- how it produces such mazes. In Section 5.3, I discuss the way we approached game testing goals and in Section 5.4, I present the experimental data we used. In Section 5.5, we analyze the results of the experiments and, in Section 5.6, we discuss the findings of these results.

We conducted the experiments using Windows 7 in an Intel® Core™ i7-2630QM CPU @ 2.00 GHz with 6 GB of RAM. The implementation of the experiments has been accomplished using the game engine Unity 3D 4.1. In order to use Lua in the environment of Unity, we used the asset LuaInterface with Lua 5.2

5.2 PERFECT MAZES WITH THE RECURSIVE BACKTRACKER ALGORITHM

The specific algorithm was chosen based on the table at the Chapter 2. The Recursive Backtracker generates perfect Mazes quite fast but not faster than others with low memory usage. But, above all, the most important characteristic is that it has the lowest dead end rate (10%) and the highest solution rate (19%). Nevertheless, the complexity of the Maze sufficiently incommodes our custom made algorithms, forcing them to cover

a large enough number of paths on their way to achieve their goal and, thus, helps us record representative results.

The difficulty is categorized based on the maze size and, thus, its complexity. Therefore, we developed a 10x10-block maze (level-1), a 15x15-block one (level-2), a 17x17-block one (level-3) and, last but not least, a 20x20-block maze (level-4).

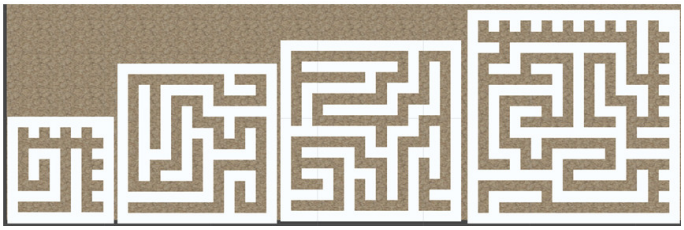

Figure 6: Maze complexity according to size (left to right) = (easy to difficult)

The mazes were used in the experiments, in which the three algorithms were run using the pseudo-DSL. The varying difficulty had an obvious impact on the results, especially on the time elapsed and the number of decisions that were made before the goal was accomplished.

The logic behind the development of each maze follows the Algorithm 5 that is mostly used in solving as well as developing such mazes.

In our case, a maze is represented by a network of cells 1x1x1 m^3. The surrounding cells of each cell are recognized as neighbors. Additionally, each cell can be a wall or a path and during the execution of the algorithm every cell visited cell is stored in an array.

```
Make the initial cell the current cell and mark it as visited;
while there are unvisited cells do
    if the current cell has any neighbors which have not been visited then
        Choose randomly one of the unvisited neighbors;
        Push the chosen cell to the stack;
        Remove the wall between the current cell and the chosen cell;
        Make the chosen cell the current cell and mark it as visited;
    else
        Pop a cell from the stack;
        Make it the current cell;
    end
end
```

Algorithm 5: Recursive backtracker algorithm

As I mentioned in Chapter 2, a perfect maze only has one solution to the problem and no hidden/faulty exits.

5.3 EXPRESSING TESTING GOALS TO GAME TERMS

For the sake of the experiments, we obviously need a game. Thus, we decided to implement a basic 3D shooting game and pass it to System Under Test (SUT) in order to prove that our pseudo-DSL, which was described in Chapter 4, could express game goals with respect of game terms.

To achieve that, the three implemented algorithms, namely 'random walk', 'heuristic AI walk' and 'evowalk', were used to test the system. Suppose we have a 3D dimensional controlled environment, one gameObject-player, one gameObject-target and one goal. The main goal of the game is to locate the gameObject-target and shoot it, using the input controls of the gameObject-player. The term controlled 3D environment implies a 3D environment the difficulty of which can be measured via controlled variables, such as the height and width of the game environment. For that purpose, we used mazes which we implemented by using the Recursive Backtracker algorithm, as described in Section 5.2.

A game goal's complexity varies, depending on the number of parameters of which the current state consists of and the given freedom to use these parameters in order to express that goal with regard to game terms. That also means that a goal can be expressed in multiple ways. In our case, the game goals are quite simple and can be expressed with one condition, which checks the distance between the gameObject-player and the gameObject-target in every state. If the distance is below a specified value then the goal has been achieved. In order to achieve that goal, a Lua script will control the gameObject-player while constantly checking the condition of the goal. For example, if the current goal is to approach the target and not eliminate it, it is expressed by :

$$If(|whereis(Player) - whereis(Target)| <= 5)\ then\ break\ endif$$

On the other hand, if the goal was to eliminate the gameObject-target, the condition would check the amount of targets in the state.

The same functionality can be used to check correctness condition at the runtime. In our case, the assertion that the position of the gameObject-player is always on the terrain was added.

$$Assert((Player.Transform.position.y > 1)\&\&(Player.Transform.position.y < 4))$$

The first part of that condition asserts that the gameObject-player is always over the terrain, while the second part of the condition asserts that the gameObject-player does not jump over walls.Such an assertion is interpreted as a game invariant, which should hold on every frame update.

Furthermore, for the sake of debugging, a bug reporting functionality was added; it logs and updates -in real time- the console with bugs found during the test. In this experiment, we will also test for bugs that violate the asserted condition where the player jumps over a wall. Of course, in practice, we would have to check more than one assertion. Therefore, each action of the controlled gameObject is considered as a sub-goal. The assertion can be achieved by asserting a statement immediately after an action or running a different coroutine at the same time.

The above examples of assertions actually express visual constrains on the game. So, it is at least possible to express them in our pseudo-DSL. A violation corresponds to a visual bug, and we will see if our testing algorithms can actually catch such a bug.

5.4 EXPERIMENTAL DATA

Before analyzing the results of the experiments, let us first present the parameters which were used to configure the System Under Test. As mentioned before, each state of the SUT must be fully controllable, so that we are able to measure and compare every element of the game.

For the graphical interface of the game, we used the four perfect mazes in Figure 6, of four levels of difficulty. Table 7 lists the exact size of these mazes, according to the difficulty level.

Difficulty	Width	Height
level-1	10	10
level-2	15	15
level-3	17	17
level-4	20	20

Table 7: Size of the maze compared to the difficulty level

.

Before the experiments are run, cluster sampling was performed from a sample of 100 runs of each testing algorithm for the level-1. Ten data groups indicated the average number of experiments that is needed to reach fairly representative results, which was

really convenient, given the limited time. That resulted in running ten experiments for each testing algorithm, for each level; or, according to Unity terminology, for each scene.

For each experiment we accomplished, we measured the duration of each execution of the algorithm; the amount of bugs that were found during the test; whether the goal was achieved or not; the number of the decisions the gameObject-player took during the execution; and the distance between the gameObject-player from the gameObject-target. The gameObject-player speed and scale were measured, as well, for debugging purposes. Raising the speed or the scale of the player could result in bugs deliberately.

5.5 RESULTS

In the following chapter, we present the results of each algorithm that we tested. The results of the experiments are presented in box plots which show the relation between the decisions made during the execution of the algorithm or the time needed to run the algorithm and the difficulty of the maze. More specifically, a box plot displays the range and distribution of data along a number line. The full reports of the testing algorithms can be found in the *APPENDIX*.

Furthermore, it is worth mentioning that during the experiments we located and solved plenty of visual and programming bugs with the help of our DSL. These bugs were not planted on purpose and their solution helped to properly continue with the experiments.

One of the most important bugs, which was of course solved, was the continuously increasing speed of the gameObject-player during the tests. The bug was "hiding" behind the logic of the testing script which was not terminating the coroutines of each movement function. That resulted in the gameObject-player's speed multiplying and allowing the player to pass or jump over the maze walls.

5.5.1 *Random walk results*

A random walk algorithm will not provide efficient solutions but it will cover bigger paths by taking more decisions. The term decision stands for the movement direction on every state. Furthermore, it is expected that the number of decisions for every second will be enormous. Therefore, we also test whether our pseudo-DSL can cope with a stressed environment or not.

The following figure shows example of the relation between the number of decisions made and the duration (in seconds) for 10 runs of the same testing algorithm, at the

level-1. It is quite clear that at the fourth run of the random walk algorithm the gameObject-player trapped for a while between walls which resulted to a great amount of decisions.

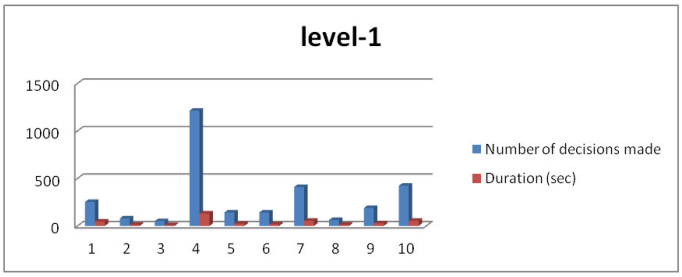

Figure 7: Relation between the number of decisions made and the duration (in seconds) for 10 runs of the same testing algorithm

The Figure 8 presents the relation between the difficulty of the maze and the average duration of the SUT's run in order to achieve the specified goal. The vertical axis represents the duration in seconds per run with the maximum limit of one hour or 3600 seconds. The horizontal axis shows the difficulty level for each maze.

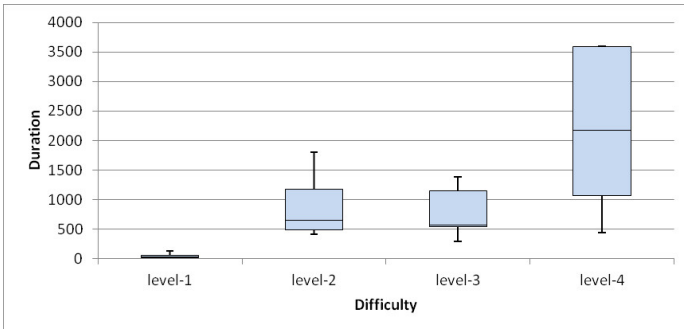

Figure 8: Relation between the difficulty of the maze and the duration

Looking at the plots from left to right, it is clear there is an abnormality which is caused by the randomness of the algorithm. The two whiskers, or the first and the last quartile, of each plot show the minimum and the maximum time spent, while the middle line defines the median time. Reaching the limit of 3600 seconds implies the fact that the algorithm could not solve the maze and achieve the goal, either by blocking itself in a corner or by producing a bug. In the case of the last box plot, the algorithm did not manage to solve the maze and spent the available time blocked in a corner.

The first plot of the next figure presents our first attempt to test level-1 reducing the scale of the gameObject-player to 50% which can be compared with the second plot where the scale of the gameObject-player is normal (scale:1). By reducing the scale of the gameObject-player, we reduce its volume in the 3D environment. It is well noticeable that a smaller gameObject-player is forced to cover more space by taking more decisions.

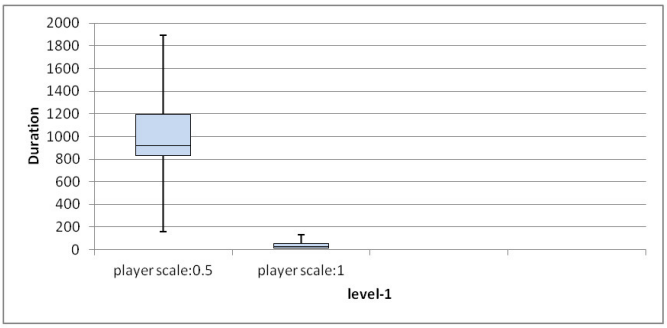

Figure 9: Relation between player scale and duration

The Figure 9 suggests that there is a relation between the execution time of the algorithm and the parameters of the environment.

The Figure 10 presents the relation between the difficulty of the maze and the number of decisions made to achieve the specified goal on each run of the SUT.

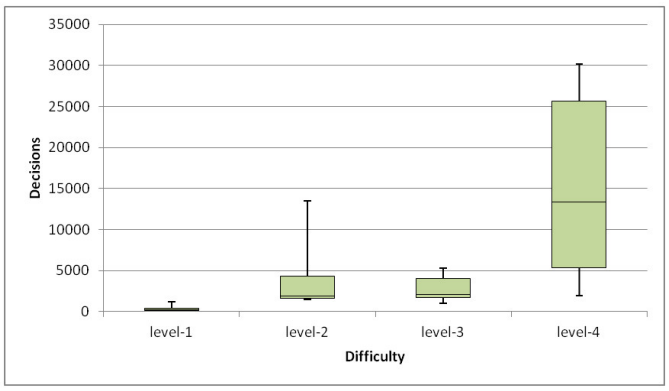

Figure 10: Relation between the difficulty of the maze and the decisions made

Comparing the whiskers of Figure 8 and Figure 10, we may observe the proportion between the decisions taken and the execution time on an average rate of 6 decisions per

second.

5.5.2 *Heuristic-AI walk results*

The custom made heuristic-AI algorithm will try to solve the specific goal following logical decisions according to the known position of the target goal. The fact that the gameObject-player will always try to avoid unimportant obstacles and always try to approach the target results to an enormous amount of decisions in a short period of time. A characteristic example of the previous statement can be observed in the next figure which shows the number of decisions made and the duration of the algorithm, for 10 runs in a maze level-2. It is calculated that there are made almost 55 decisions per second in total average.

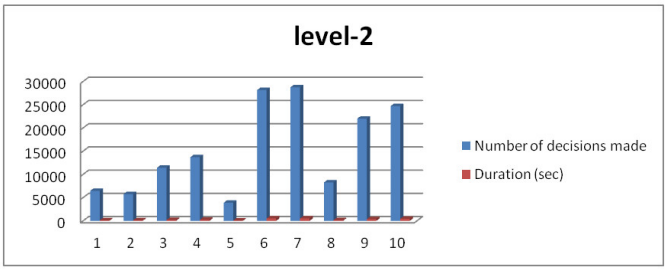

Figure 11: Relation between the number of decisions made and the duration (in seconds) for 10 runs of the same testing algorithm

The next two box plots can be considered as the most important figures in my research. The scaling of the time and decisions according to the level of difficulty can be observed in the blue box of each plot.

Figure 12 presents the relation between the duration and the difficulty of the maze. The whiskers, of the first and third plot, imply that the algorithm failed to complete the goal at least once which was caused by either finding a bug or losing its way to the target position. The last plot indicates that the algorithm was not able to solve the maze in the given time (3600 seconds) by 50%. The fact that the algorithm could solve for only the 50% of the iterations can be observed by the maximum and the second quadrille's position which is the same in both cases.

Figure 13 verifies the previous observations which are the two peaks of the first and third plot, showing the very large amount of decisions and more importantly the gradually

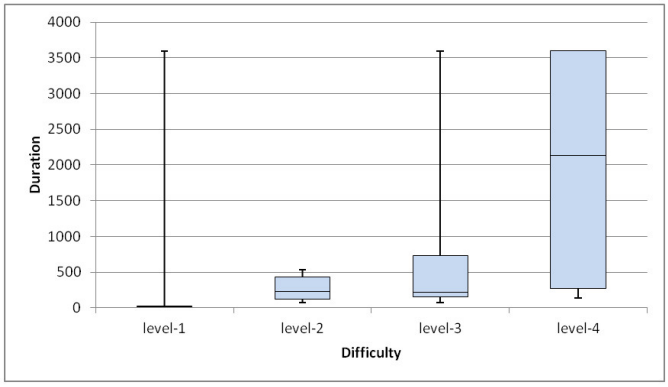

Figure 12: Relation between the difficulty of the maze and the duration

scaling ability of the algorithm.

The maximum of the third box plot indicates the fact that there is possibility of a bug, which is actually true. At the first iteration of the algorithm in the maze with at level-3, the algorithm produced a bug by making 51032 decisions in 892 seconds, almost 57 decisions per second, forcing the gameObject-player to jump over a wall and take a "shortcut" to the goal destination. As mentioned at Chapter 5.3, a bug can be detected and reported, if the assertion is violated.

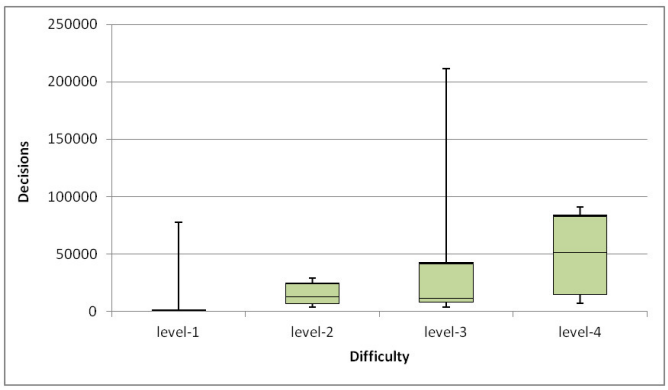

Figure 13: Relation between the difficulty of the maze and the decisions made

It is worth noticing that the "heuristic-AI walk" is the fastest algorithm among those we tested in our experiments. Furthermore, it should be noted that the algorithm is intentionally stressing the gameObject-player to produce a bug in the SUT by driving the gameObject-player to go towards the known gameObject-target position even if there

is a wall blocking its way.

5.5.3 *Evowalk results*

The most typical attribute of the evowalk algorithm is the deterministic and low amount of decisions made for each iteration (see Figure 16). Furthermore, the fact that the algorithm starts with a list of random decisions and learns how to locate the target during the procedure implies that the duration till the accomplishment of the goal varies (see Figure 14).

Moreover, it is important to state that the first two difficulty levels of the current algorithm were tested differently from the last two which is also viewable in Figure 14. In the case of level-1 and level-2, the evowalk algorithm can solve the puzzle and complete the goal in a short period of time by 90% success. On the other hand, at level-3 and level-4, the algorithm cannot achieve the goal even after ten hours.

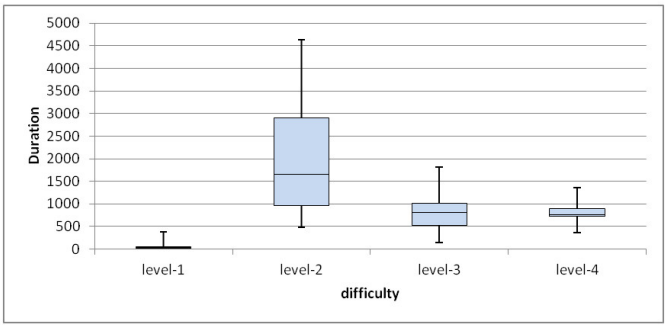

Figure 14: Relation between the difficulty of the maze and the duration

The logical explanation behind this problematic reaction is that the evowalk produces a deterministic amount of movements which are completely random during the first iteration. For example, the gameObject-player might approach the gameObject-target very closely at the first half of their movement procedure but immediately turn back to their starting position. The cost of the specific DNA is calculated at the end of the execution, producing a false result.

In order to solve this misbehavior, I have added certain sub goals in vital places that serve as milestones for the algorithm. Using sub goals, the algorithm can define the sub goal position as a starting point -after reaching it for the first time- and start from that during the next iteration. Of course, following the specified method for the last two

runs, the evowalk algorithm stops behaving as a genetic algorithm and reacts more like a random one.

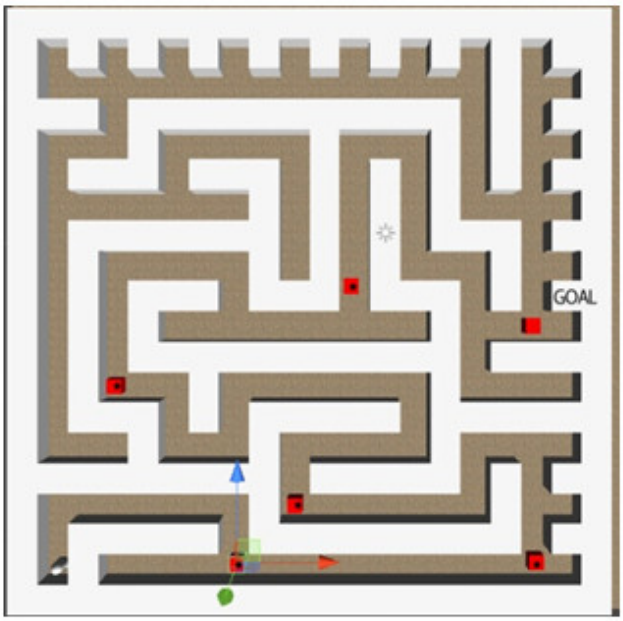

Figure 15: The player is the white object on the bottom left. Red cubes with the black bullet are the sub goals. The red cube is the Goal

According to Figure 14, observing the whiskers of the last two plots where our customized approach is used, the evowalk algorithm not only managed to solve the goal but also accomplished it quite fast. Furthermore, an important fact that arises from the comparison of the decision box plots of each algorithm and level of difficulty is that the total number of decisions made by the evowalk algorithm is less than the lower whisker of the other plots.

The box plots of Figure 16 present the range of the decisions made which are really low and verify the previous fact.

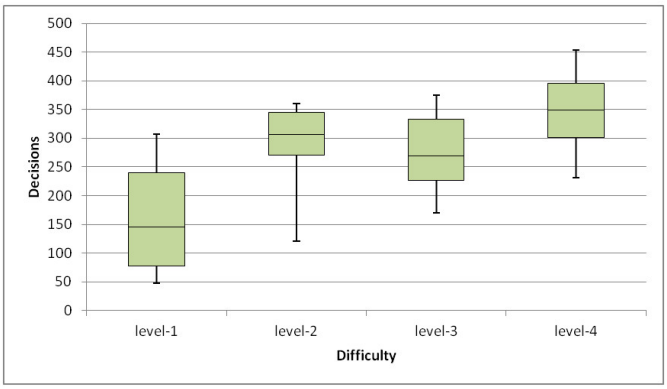

Figure 16: Relation between the difficulty of the maze and the decisions made

5.6 DISCUSSION

Taking into consideration the Related Work from Chapter 3 and our pseudo-DSL from Chapter 4, we can answer the first research question; but, in order to prove it, the results of the experiments were necessary. According to the plots of each experiment, from section 5.5, all of the test approaches managed to achieve the specified goal most of the times. The total average of the duration for each algorithm is in a predefined time range of 3600 seconds. Moreover, a negative result is usually produced when one of the tested algorithms locates a bug or gets blocked by a wall.

Furthermore, the variation of the tested algorithms implies the high usability of our pseudo-DSL, concerning the fact that it can be applied with different ways according to the tester's judgment. More customization was needed at the last algorithm we tested where we firstly introduced sub-goals. In order to solve the maze, the updated evowalk algorithm tried to solve a list of sub-goals first, saving its state every time it would achieve a sub-goal, and later aimed for the main goal. For this reason, every sub-goal can be considered equally as a goal.

To conclude, by observing the results of the experiments, we prove that a DSL and more specifically our pseudo-DSL can not only express game-related testing goals in game terms, but also break down a game testing goal into sub-goals. Our pseudo-DSL can be customized according to the specified domain needs, providing a high success rate for the specified goal. Additionally, the positive results of each level of difficulty from the plots of section 5.5 prove the fact that our pseudo-DSL can scale up according to the specified domain.

6 CONCLUSION AND FUTURE WORK

6.1 CONTEXT

In this chapter, I will present the conclusions of my research and discuss the results of my experiments, in order to try to give a straightforward answer to each and every one of the research questions, as formulated in Chapter 1.

Last but not least, I will suggest potential future work related to both my research and the Domain Specific Language that was introduced for the purpose of the thesis.

6.2 EXPRESSING GOALS AND SUB-GOALS IN GAME TERMS

With regards to the first question, the related work that is presented in Chapter 2 implies that one of the most common ways of expressing game-related testing goals in game terms could be achieved with the use of a Domain Specific Language. Even though no work of the presented bibliography contains a proper testing DSL for three-dimensional games, it should be noted that all of them use a Domain Specific Language which expresses game terms in terms of higher level concepts. The results of our experiments prove that it is also possible to express testing goals in a 3D environment by using the related pseudo-DSL.

Hence, we propose a prototype of Game Testing Pseudo Domain Specific Language (GTP-DSL) which can express game testing goals to game terms. Provided that each action of the domain is considered as a sub-goal, we can assume that each main goal can be a combination of game terms. The current fact answers the following research sub-questions:

Is it possible to break down a game testing goal into sub-goals?

According to [29], Pseudo code is an informal high-level description of the operating principle of a computer program or other algorithm. The term 'Pseudo DSL' can be attributed to the fact that the proposed GTP-DSL is a combination of the Lua syntax and the methods of the specified domain.

Furthermore, the results of the Experiments presented in Chapter 5 demonstrate the ability of the testing algorithms, which were written in GTP-DSL, to achieve the given goal. The box plots did not only prove the usability of the pseudo domain specific

language but also the scalability in terms of difficulty. In addition, GTP-DSL can also be considered as a technique with a framework that is adjustable to any given SUT which answers another research sub-question:

Is it possible to have a DSL for that purpose, and could it be generic for all types of games?

6.3 AUTOMATED GAME TESTING AND DSLS

As already mentioned in Chapter 4, the luxury of using the syntax of Lua in order to use GTP-DSL fulfilled an extra purpose, which is the automation of the test cases. While LuaInterface provides the communication between the programming language Lua and the game engine, GTP-DSL provides the communication with the SUT. That way, the test cases can be generated on the fly, according to the given instructions which are written in a Lua script, using pseudo-DSL; and regression testing can be achieved by reusing old test cases.

Furthermore, the last testing algorithm -which uses genetic algorithm techniques as described in Chapter 4- is a tangible example of the automation testing experience. The current algorithm not only generates the test cases of the SUT automatically but also takes decisions for the generation of the next test case.

I could not locate relevant work concerning the research sub-question:

Are there other kinds of DSLs that can be used for goal-oriented game testing?

As far as the DSLs I researched and studied are concerned, they have only been used for game development and particularly for two dimensional logic and turned based games.

Last but not least, it is true that many games involve randomness, concurrency, or multiple players, contributing to a high degree of non-determinism. Finding ways to solve testing goals under such conditions requires way more time than we had available. However, through the experience I have gained while working on my thesis and considering that every task or action in a SUT can be considered as a goal, I strongly believe that non-determinism is not affecting the given goal.

6.4 FUTURE WORK

6.4.1 *Reduce first, debug later*

One of the advantages of GTP-DSL is that a test case can be stored into a Lua script and re-used later, but in some cases which produce game bugs, the number of events is huge and a back-tracing utility is not implemented. In order to reduce to the minimum number of events that reproduce a failure, I propose the implementation of the delta debugging minimization algorithm ddMin. The Delta Debugging algorithm isolates failure causes automatically, by systematically narrowing down failure-inducing circumstances until the minimal set remains[30].

The implementation can be achieved by following the guidelines of the recent paper "Reduce first, debug later" [31], which proposes the use of two algorithms, ddmin and rddmin. Therefore, each failure could be minimized with the standard algorithm (ddmin), and then the minimization procedure can be conducted with rddmin (Minimizing Delta Debugging Algorithm Complemented with Reduction).

6.4.2 *Extending the evowalk algorithm*

In order to test the validity, the usability and the scalability of the GTP-DSL, we experimented with three algorithms which are described in 'Methodology', in Chapter 4, and analyzed in Chapter 5. The last algorithm, namely the "evowalk algorithm", can be extended and generalized in a way that could afford multiple keywords and test more complicated cases than just the movement of a player. Currently, the evowalk algorithm produces test cases (population) which contain a limited amount of movement keywords such as: move left, move right, move up, move down.

Furthermore, the evowalk generates test cases automatically which makes it more convenient for testing bigger domains in the future. The extension of the evowalk algorithm can be assisted by the ddmin algorithm which can result in a concrete game testing solution. Last but not least, the evowalk algorithm uses a helper function for the generation of the test cases, while the core functionality can remain untouched.

6.4.3 *Other Future Work*

Additionally, other prospects of future work can be the following:

- *A Back-end Graphical User Interface which can relate game methods to keywords for the GTP-DSL. Currently, the GTP-DSL can be connected with functions and methods of the game engine by defining them in a C# file. That way, GTP-DSL could be used by testers with no programming background.*
- *Right now, our implementation is not a standalone asset for Unity3d and requires customization for each domain. An important asset for the Game Community could result from our implementation, in order to help game developers to not only test their domain and simulate object movements, but also tie up procedures to gameObjects for their game (NPCs, enemies, objects in the scene).*

7 REFERENCES

7.1 BIBLIOGRAPHY

BIBLIOGRAPHY

[1] I. C. Mark Buenen, Makarand Teje, ``World quality report 2014,'' 2014. World Quality Report 2014.

[2] A. van Deursen, P. Klint, and J. Visser, ``Domain-specific languages: An annotated bibliography,'' *SIGPLAN Not.*, vol. 35, pp. 26--36, June 2000.

[3] Planning and artificial intelligence, 2014. http://en.wikipedia.org/wiki/Graphical_-user_interface_testing.

[4] ``Ieee standard glossary of software engineering terminology,'' *IEEE Std 610.12-1990*, pp. 1--84, Dec 1990.

[5] L. Williams, ``Testing overview and black-box testing techniques.'' unpublished paper, 2006.

[6] DSL, 2014. http://en.wikipedia.org/wiki/Domain-specific_language.

[7] A. van Deursen, P. Klint, and J. Visser, ``Domain-specific languages: An annotated bibliography,'' *SIGPLAN Not.*, vol. 35, pp. 26--36, June 2000.

[8] J. Dobbe, ``A domain-specific language for computer games,'' Master's thesis, TUDelft, 2006.

[9] A. Lua, 2014. http://www.lua.org/about.html.

[10] R. Ierusalimschy, 2014. http://lua-users.org/lists/lua-l/2007-11/msg00248.html.

[11] A. is not a 4-letter word, 2014. http://www.jamisbuck.org/presentations/ruby-conf2011/.

[12] T. L. M. Classification, 2014. http://www.astrolog.org/labyrnth/algrithm.htm.

[13] E. Algorithm, 2014. http://en.wikipedia.org/wiki/Evolutionary_algorithm.

[14] P. McMinn, ``Search-based software test data generation: A survey: Research articles,'' *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105--156, June 2004.

[15] DSEL, 2014. http://en.wikipedia.org/wiki/Domain-specific_entertainment_language.

[16] G. G. Playing, 2014. http://en.wikipedia.org/wiki/General_Game_Playing.

[17] G. G. Playing, 2014. http://www.general-game-playing.de/.

[18] M. G. Nathaniel Love, Timothy Hinrichs, ``General game playing: Game description language specification,'' tech. rep., Stanford Logic Group, Stanford, CA 94305, 2006.

[19] M. Thielscher, ``A general game description language for incomplete information games.'' School of Computer Science and Engineering The University of New South Wales, Australia, mit@cse.unsw.edu.au, 2010.

[20] S-expression, 2014. http://www.territorioscuola.com/wikipedia/en.wikipedia.php?title=S-expression.

[21] L. Hayes, *The Automated Testing Handbook*. Software Testing Institute, 1995.

[22] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[23] A. van Deursen, P. Klint, and J. Visser, ``Domain-specific languages: An annotated bibliography,'' *SIGPLAN Not.*, vol. 35, pp. 26--36, June 2000.

[24] P. code, 2014. http://en.wikipedia.org/wiki/Pseudocode.

[25] E. style language, 2014. http://en.wikipedia.org/wiki/Embedded_style_language.

[26] I. game Console, 2014. https://github.com/mikelovesrobots/unity3d-console.

[27] L. M. Introduction, 2014. http://www.lua.org/manual/5.2/manual.html.

[28] F. Mascarenhas, *LuaInterface: User's Guide*. 1Departamento de Inform´atica, PUC-Rio Rua Marqu^es de S˜ao Vicente, Rio de Janeiro, RJ, Brasil.

[29] P. code, 2014. http://en.wikipedia.org/wiki/Pseudocode.

[30] D. Debugging, 2014. http://en.wikipedia.org/wiki/Delta_Debugging.

[31] A. Elyasov, W. Prasetya, J. Hage, and A. Nikas, ``Reduce first, debug later,'' in *Proceedings of the 9th International Workshop on Automation of Software Test*, AST 2014, (New York, NY, USA), pp. 57--63, ACM, 2014.

8

APPENDIX

8.1 CONTEXT

In this Chapter, you will find terminology which was used in the current research thesis, code examples and experiment reports.

8.2 EXPERIMENT REPORTS

In the following Section, we present the full reports of our experiments for each algorithm that we tested. The maximum time for each iteration is 3600 seconds and the assertion which is checked on every frame is:
$(Player.Transform.position.y > 1)$ && $(Player.Transform.position.y < 4)$

8.2.1 *Random Walk Algorithm full results*

Difficulty : level-1 (Maze 10x10) with player.scale=0.5

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	730	0	1	4413	8	0.5	2.558213
2	1646	0	1	16890	8	0.5	1.663525
3	826	0	1	5831	8	0.5	2.256318
4	1243	0	1	14438	8	0.5	2.38112
5	159	0	1	1929	8	0.5	2.21715
6	847	0	1	6525	8	0.5	2.248209
7	865	0	1	7503	8	0.5	2.266233
8	1891	0	1	13984	8	0.5	1.499887
9	977	0	1	9283	8	0.5	2.2775
10	1037	0	1	9805	8	0.5	2.273067

Difficulty : level-1 (Maze 10x10) with player.scale=1

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale
1	47	0	1	252	8	1
2	12	0	1	79	8	1
3	9	0	1	50	8	1
4	131	0	1	1214	8	1
5	24	0	1	142	8	1
6	17	0	1	142	8	1
7	54	0	1	410	8	1
8	13	0	1	62	8	1
9	28	0	1	188	8	1
10	54	0	1	423	8	1

Difficulty : level-2 (Maze 15x15)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	1808	0	1	13497	8	1	2.318712
2	1224	0	1	4543	8	1	2.41458
3	488	0	1	1501	8	1	2.348471
4	550	0	1	1705	8	1	1.99696
5	484	0	1	1810	8	1	2.01847
6	1403	0	1	5604	8	1	2.14816
7	759	0	1	2002	8	1	1.99788
8	478	0	1	1448	8	1	2.34441
9	419	0	1	1592	8	1	2.462137
10	1049	0	1	3644	8	1	2.432867

8.2.2 *Random Walk Algorithm full results*

Difficulty : level-3 (Maze 17x17)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	540	0	1	2043	8	1	1.518213
2	1354	0	1	4826	8	1	1.11438
3	295	0	1	963	8	1	1.141473
4	542	0	1	2121	8	1	1.81612
5	1391	0	1	5272	8	1	2.23547
6	574	0	1	1770	8	1	2.12476
7	1321	0	1	4605	8	1	1.47110
8	645	0	1	2279	8	1	2.30750
9	354	0	1	1273	8	1	2.35421
10	555	0	1	1641	8	1	1.43286

Difficulty : level-4 (Maze 20x20)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	1652	0	1	6928	8	1	2.48713
2	3550	0	1	27384	8	1	2.34417
3	3600	0	0	23087	8	1	18.17718
4	3600	0	0	26474	8	1	23.24874
5	2700	0	1	19821	8	1	2.17445
6	1327	0	1	6031	8	1	1.47448
7	3600	0	0	30154	8	1	19.0157
8	987	0	1	5121	8	1	1.57118
9	440	0	1	1925	8	1	1.47451
10	502	0	1	2404	8	1	1.34847

8.2.3 *Heuristic AI Algorithm full results*

Difficulty : level-1 (Maze 10x10)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	7	0	1	225	8	1	1.420209
2	11	0	1	361	8	1	1.355443
3	3600	0	0	77785	8	1	11.39836
4	28	0	1	838	8	1	1.377096
5	50	0	1	1584	8	1	1.461822
6	28	0	1	845	8	1	1.36695
7	18	0	1	569	8	1	1.328511
8	8	0	1	275	8	1	1.398472
9	7	0	1	240	8	1	1.458898
10	21	0	1	648	8	1	1.347177

Difficulty : level-2 (Maze 15x15)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	117	0	1	6468	8	1	1.374443
2	101	0	1	5779	8	1	1.175977
3	207	0	1	11430	8	1	1.404168
4	248	0	1	13718	8	1	1.423365
5	69	0	1	3907	8	1	1.460547
6	538	0	1	28159	8	1	1.492137
7	535	0	1	28766	8	1	0.876626
8	149	0	1	8270	8	1	1.432229
9	382	0	1	22007	8	1	0.844346
10	451	0	1	24717	8	1	1.26811

Difficulty : level-3 (Maze 17x17)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	892	1(out of map)	0	51032	8	1	21.5143
2	78	0	1	4176	8	1	1.241423
3	68	0	1	3538	8	1	1.972894
4	193	0	1	10519	8	1	1.490372
5	191	0	1	10185	8	1	1.25089
6	3600	0	0	85179	8	1	21.78111
7	237	0	1	12723	8	1	1.441642
8	141	0	1	7522	8	1	1.43584
9	3600	0	0	211791	8	1	17.1871
10	255	0	1	13748	8	1	1.25475

8.2.4 *Heuristic AI Algorithm full results*

Difficulty : level-4 (Maze 20x20)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target
1	183	0	1	10322	8	1	1.36314
2	3600	0	0	89248	8	1	15.57478
3	201	0	1	10758	8	1	1.476289
4	462	0	1	26214	8	1	1.154065
5	3600	0	0	77174	8	1	14.98474
6	3600	0	0	84741	8	1	15.17749
7	665	0	1	26995	8	1	1.423806
8	3600	0	0	75547	8	1	16.54779
9	141	0	1	7432	8	1	1.491711
10	3600	0	0	91271	8	1	8.21571

8.2.5 *Evowalk Algorithm full results*

Difficulty : level-1 (Maze 10x10)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target	Total time	DNA no	Iterations
1	23	0	1	114	8	1	1.54264	23	1	1
2	47	0	1	236	8	1	1.48871	227	2	1
3	10	0	1	47	8	1	1.57884	10	1	1
4	27	0	1	177	8	1	1.47713	27	1	1
5	15	0	1	75	8	1	1.5741	15	1	1
6	10	1 (wall)	0	52	8	1	12.9597	10	1	1
7	47	0	1	241	8	1	2.100214	47	1	1
8	59	0	1	293	8	1	1.39741	59	1	1
9	17	0	1	86	8	1	1.472144	377	3	1
10	56	0	1	307	8	1	1.584417	56	1	1

Difficulty : level-2 (Maze 15x15)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target	Total time ((DNA-1)*180+duration+(Iterations-1)*10*180)	DNA no	Iterations
1	107	0	1	232	8	1	1.54774	2267	3	2
2	131	0	1	308	8	1	1.14745	3191	8	2
3	136	0	1	355	8	1	1.14775	4636	6	3
4	45	0	1	121	8	1	1.15998	3105	8	2
5	120	0	1	261	8	1	1.98471	480	3	1
6	150	0	1	358	8	1	2.51175	870	5	1
7	132	0	1	314	8	1	2.15774	492	3	1
8	154	0	1	361	8	1	2.15476	1261	6	1
9	126	0	1	299	8	1	1.58124	1926	1	2
10	134	0	1	304	8	1	2.47841	1394	8	1

8.2.6 *Evowalk Algorithm full results*

Difficulty : level-3 (Maze 17x17)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target	Total time ((DNA-1)*60+duration+(Iterations-1)*10*60)	DNA no	Iterations
1	35	0	1	83	8	1	1.4879	815	4	2
Sub goal no			Duration					decisions		
1			36					93		
2			45					118		
3			17					51		
2	19	0	1	42	8	1	1.32187	619	1	2
Sub goal no			Duration					decisions		
1			16					33		
2			50					113		
3			41					81		
3	23	0	1	56	8	1	1.21248	1043	8	2
Sub goal no			Duration					decisions		
1			46					103		
2			35					76		
3			44					104		
4	29	0	1	72	8	1	1.3997	389	7	1
Sub goal no			Duration					decisions		
1			20					54		
2			9					33		
3			36					11		
5	20	0	1	42	8	1	1.7956	1820	1	4
Sub goal no			Duration					Decisions		
1			54					120		
2			56					118		
3			43					95		
6	23	0	1	46	8	1	1.2344	503	9	1
Sub goal no			Duration					Decisions		
1			43					101		
2			48					107		
3			39					15		
7	13	0	1	40	8	1	2.00472	793	4	2
Sub goal no			Duration					Decisions		
1			25					79		
2			40					95		
3			51					100		
8	28	0	1	65	8	1	1.1843	928	6	2
Sub goal no			Duration					decisions		
1			31					83		
2			12					45		
3			40					21		
9	25	0	1	59	8	1	1.99145	1345	3	3
Sub goal no			Duration					decisions		
1			31					92		
2			45					102		
3			34					11		
10	25	0	1	60	8	1	1.28668	145	3	1
Sub goal no			Duration					decisions		

8.2.7 *Evowalk Algorithm full results*

1	10	16
2	9	15
3	53	108

Difficulty : level-4 (Maze 20x20)

Try	Duration (seconds)	#bugs found	Goal achieved (0|1)	#decisions made	Player speed (m/s)	Player scale	Distance from target	Total time ((DNA-1)*60+duration+(Iterations-1)*10*60)	DNA no	Iterations
1	14	0	1	54	8	1	1.14374	914	6	2

Sub goal no	Duration	decisions
1	21	64
2	5	8
3	24	73
4	27	80
5	16	42

2	28	0	1	21	8	1	1.57745	868	5	2

Sub goal no	Duration	decisions
1	49	104
2	45	108
3	5	53
4	18	33
5	40	83

3	24	0	1	63	8	1	1.61751	744	3	2

Sub goal no	Duration	decisions
1	9	21
2	39	83
3	7	25
4	57	116
5	22	39

4	11	0	1	21	8	1	1.54774	371	7	1

Sub goal no	Duration	decisions
1	40	102
2	10	19
3	15	43
4	24	50
5	48	117

5	43	0	1	106	8	1	1.34748	1003	7	2

Sub goal no	Duration	decisions
1	9	18
2	30	58
3	13	47
4	20	37
5	14	28

6	25	0	1	52	8	1	1.24315	445	8	1

Sub goal no	Duration	decisions
1	12	25
2	35	62
3	11	39
4	19	31
5	10	22

7	15	0	1	34	8	1	1.35412	735	3	2

8.2.8 *Evowalk Algorithm full results*

Sub goal no	Duration	decisions
1	25	64
2	39	68
3	15	55
4	19	37
5	40	119

8	16	0	1	32	8	1	1.41127	736	3	2

Sub goal no	Duration	decisions
1	12	23
2	32	65
3	16	59
4	18	34
5	21	61

9	26	0	1	55	8	1	2.13125	766	3	2

Sub goal no	Duration	decisions
1	16	61
2	30	57
3	40	78
4	37	92
5	41	110

10	36	0	1	89	8	1	1.18237	1356	3	3

Sub goal no	Duration	decisions
1	22	61
2	36	70
3	20	51
4	31	75
5	23	61

8.3 CODE SAMPLES

In the following Section, we present code examples of the known game tic tac toe, written in GDL, ViGL and Zillions of games.

8.3.1 *GDL*

```
;;;;;;;;;;;;;;;;;;;;;;;;
;; Tictactoe
;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;
        (role x)
        (role o)
;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;
        (init (cell 1 1 b))
        (init (cell 1 2 b))
        (init (cell 1 3 b))
        (init (cell 2 1 b))
        (init (cell 2 2 b))
        (init (cell 2 3 b))
        (init (cell 3 1 b))
        (init (cell 3 2 b))
        (init (cell 3 3 b))
        (init (control x))
;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;
;; Cell
        (<= (next (cell ?x ?y ?player))
                (does ?player (mark ?x ?y)))
        (<= (next (cell ?x ?y ?mark))
                (true (cell ?x ?y ?mark))
                (does ?player (mark ?m ?n))
                (distinctCell ?x ?y ?m ?n))
;; Control
        (<= (next (control x))
                (true (control o)))
        (<= (next (control o))
                (true (control x)))
;;;;;;;;;;;;;;;;;;;;;;;;
;; Views
;;;;;;;;;;;;;;;;;;;;;;;;
        (<= (row ?x ?player)
                (true (cell ?x 1 ?player))
                (true (cell ?x 2 ?player))
                (true (cell ?x 3 ?player)))
        (<= (column ?y ?player)
                (true (cell 1 ?y ?player))
                (true (cell 2 ?y ?player))
                (true (cell 3 ?y ?player)))
        (<= (diagonal ?player)
                (true (cell 1 1 ?player))
                (true (cell 2 2 ?player))
                (true (cell 3 3 ?player)))
        (<= (diagonal ?player)
                (true (cell 1 3 ?player))
                (true (cell 2 2 ?player))
                (true (cell 3 1 ?player)))
        (<= (line ?player) (row ?x ?player))
        (<= (line ?player) (column ?y ?player))
```

8.3.2 *GDL*

```
        (<= (line ?player) (diagonal ?player))
        (<= open (true (cell ?x ?y b)))
        (<= (distinctCell ?x ?y ?m ?n) (distinct ?x ?m))
        (<= (distinctCell ?x ?y ?m ?n) (distinct ?y ?n))
;;;;;;;;;;;;;;;;;;;;;;;;;
;; Legal Moves
;;;;;;;;;;;;;;;;;;;;;;;;;
        (<= (legal ?player (mark ?x ?y))
                (true (cell ?x ?y b))
                (true (control ?player)))
        (<= (legal x noop)
                (true (control o)))
        (<= (legal o noop)
                (true (control x)))
;;;;;;;;;;;;;;;;;;;;;;;;;
;; Goals
;;;;;;;;;;;;;;;;;;;;;;;;;
        (<= (goal ?player 100)
                (line ?player))
        (<= (goal ?player 50)
                (not (line x))
                (not (line o))
                (not open))
        (<= (goal ?player1 0)
                (line ?player2)
                (distinct ?player1 ?player2))
        (<= (goal ?player 0)
                (not (line x))
                (not (line o))
                open)
;;;;;;;;;;;;;;;;;;;;;;;;;
;; Terminal
;;;;;;;;;;;;;;;;;;;;;;;;;
(<= terminal
        (line ?player))
(<= terminal
        (not open))
```

8.3.3 *ViGL*

```
<vigl resolution="300x300">
    <code location="begin">
    $TURN = true
    class GameObject
        def inside?(mouseState)
            if(mouseState.button==Mouse::BUTTON_LEFT and
                mouseState.x >= @renderable_object.rectangle_shape.point.x and
                mouseState.y >= @renderable_object.rectangle_shape.point.y and
                mouseState.x < @renderable_object.rectangle_shape.point.x +
                    @renderable_object.rectangle_shape.dimensions.width and
                mouseState.y < @renderable_object.rectangle_shape.point.y +
                    @renderable_object.rectangle_shape.dimensions.height
            )
                true
            else
                false
            end
        end
    end
    </code>
    <objectdef name="Square">
        <shape>
            <square length="100" />
            <graphics border="#000000" color="#ffffff"/>
        </shape>
        <actions>
            <method action="onMouseDown(state)">
                <code>
                    if(inside? state and
@renderable_object.renderable_properties.fill == Color::WHITE)
                        @renderable_object.renderable_properties.fill =
                        if $TURN
                            Color::RED
                        else
                            Color::BLUE
                        end
                        $TURN = !$TURN
                    end
                </code>
            </method>
        </actions>
    </objectdef>

    <world>
        <object parent="Square"><shape><square point="0,0" /></shape></object>
        <object parent="Square"><shape><square point="100,0" /></shape></object>
        <object parent="Square"><shape><square point="200,0" /></shape></object>
        <object parent="Square"><shape><square point="0,100" /></shape></object>
        <object parent="Square"><shape><square point="100,100" /></shape></object>
        <object parent="Square"><shape><square point="200,100" /></shape></object>
        <object parent="Square"><shape><square point="0,200" /></shape></object>
        <object parent="Square"><shape><square point="100,200" /></shape></object>
        <object parent="Square"><shape><square point="200,200" /></shape></object>
    </world>
</vigl>
```

8.3.4 *Zillions of Games*

```
(game
        (title "Tic-Tac-Toe")
        (description "One side takes X's and the other side takes O's.
                Players alternate placing their marks on open spots.
                The object is to get three of your marks in a row horizontally,
                vertically, or diagonally. If neither side accomplishes this,
                it's a cat's game (a draw).")
        (history "Tic-Tac-Toe was an old adaptation of Three Men's Morris to
                situations where there were no available pieces. You can draw or
                carve marks and they are never moved. It is played all over the
                world under various names, such as 'Noughts and Crosses' in
                England.")
        (strategy "With perfect play, Tic-Tac-Toe is a draw. Against less
                than perfect opponents it's an advantage to go first, as having an
                extra mark on the board never hurts your position. The center is
                the key square as 4 possible wins go through it. The corners are
                next best as 3 wins go through each of them. The remaining
                squares are least valuable, as only 2 wins go through them.
                Try to get in positions where you can 'trap' your opponent by
                threatening two 3-in-a-rows simultaneously with a single move. To
                be a good player, you must not only know how to draw as the second
                player, you must also be able to takes advantage of bad play.")
        (players X O)
        (turn-order X O)
        (board
                (image "images\TicTacToe\TTTbrd.bmp")
                (grid
                        (start-rectangle 16 16 112 112) ; top-left position
                        (dimensions ;3x3
                                ("top-/middle-/bottom-" (0 112)) ; rows
                                ("left/middle/right" (112 0))) ; columns
                        (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
                )
        )
        (piece
                (name man)
                (help "Man: drops on any empty square")
                (image X "images\TicTacToe\TTTX.bmp"
                        O "images\TicTacToe\TTTO.bmp")
                (drops ((verify empty?) add))
        )
        (board-setup
                (X (man off 5))
                (O (man off 5))
        )
        (draw-condition (X O) stalemated)
        (win-condition (X O)
                (or (relative-config man n man n man)
                        (relative-config man e man e man)
                        (relative-config man ne man ne man)
                        (relative-config man nw man nw man)
                )
        )
)
```