Automated Testing in Virtual Worlds

Dimitrios Loubos, 5691982

A thesis presented for the degree of
Game and Media Technology Msc.

Game and Media Technology
Utrecht University
Netherlands
March 2018

Acknowledgements

I would first like to thank my thesis supervisor, Mr. I.S.W.B. Prasetya for his guidance and support during the research. His door was always open when I needed any help and his insight steered me the right way on completing my research.

Many thanks to my second supervisor Mr. Frank Dignum for his valuable feedback and evaluation of my thesis.

I would also like to thank my family for supporting me throughout my years of study.

Automated Testing in Virtual Worlds

Master Thesis

Dimitrios Loubos
Utrecht University
d.loubos@students.uu.nl

ABSTRACT

In this study we train an agent in the video game Minecraft, to gather items in the world space and combine them to automatically test the functionality of the crafting system of the game. We are inspired by the ideas of reinforcement learning to teach the agent a certain behavior based solely on previous experience, while we test his ability to correctly predict the exact recipe for crafting specific items. We perform experiments with different test cases to explore the success of such process and the time needed to test the crafting feature. Keeping in mind that our main purpose is to explore functional testing in virtual worlds, we conclude that such an automation process can certainly ease the workload of testing.

KEYWORDS

Functional Testing, Automated Testing, Reinforcement Learning

ACM Reference Format:
Dimitrios Loubos. 2018. Automated Testing in Virtual Worlds: Master Thesis. In *Proceedings of Master Thesis.* ACM, New York, NY, USA, Article 4, 18 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Recently, the rise of low-cost, high performance home computing and subsequent growth of the computer games industry has led to the emergence of interactive video games with the creation of entirely new forms of game-play and corresponding genres of games.

Video games make use of increasingly complex and detailed virtual environments, often incorporating human controlled protagonists and many computer controlled opponents (referred to as game agents and opponent agents, respectively). These agents require an ever more realistic and believable set of behaviours for a game engine's AI to generate.

With a wide range of potential objectives and features associated with game-play, as indicated by the plethora of genres arising from video games, such as "First-Person Shooter Games", "Action-Adventure Games", "Role-Playing Games", "Strategy Games", "Simulation Games" and "Sports Games", the specifications and requirements for a game AI typically include both genre-specific goals and general low-level behaviours. By utilizing a percentage of the overall computational resources used within a game, the game AI facilitates the autonomous selection of behaviours for game agents through game specific perception, navigation and decision making sub-systems[11].

Old 2D platform games had fewer rules, significantly smaller action space and simpler gameplay thus making the AI problems more trivial for researchers to provide a "smart" AI agent. However, for more complex games it is harder to provide a good AI agent, and writing a new game playing agent for each new game would make the process more time consuming. Furthermore, a single hand-crafted agent may be blind to novel aspects of evolved gameplay elements that the designer of the AI agent had not considered [18].

So far the there doesn't exist the perfect generalized method of AI that solves the problems above. Each game, requires its own AI approach according to its features, gameplay, path planning and so on. Thus researchers try to combine game design patterns with current state of the art AI techniques to create new AI based games [30].

Extensive testing has to be done in order to assure the quality of a game. From bigger gaming companies [5] to smaller ones, from platform games to computer and mobile games [13], testing plays an important role in the software development process. However 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.

Test automation increases the overall software efficiency and ensures robust software quality. In addition it accelerates the testing procedure allowing for the testing to be carried out repeatedly, delivering faster results each time with lesser effort and time. As a consequence it accelerates the development life cycle of the product. Furthermore, due to the repetitive nature of test automation, test cases are reusable and can hence be utilized through different approaches. Finally it may reduce the costs of a company as they don't require to hire human testers to perform the testing phase of the software.

Nowadays, almost all of the modern gaming companies use game engines, which are state-of-the-art resources that integrate both AI techniques and testing/performance tools for the developers to use. However, these game engines are usually commercial software that only game engineers use. There exist a few free game engines like Unity and Unreal Engine, but those require extensive knowledge in order to be fully useful for game development.

However, there has recently been developed a few novel AI software like Malmo Project, Deep Mind and Open Universe AI that allows for anyone to build an agent and test the AI in more complex environments in real time. Our main focus will cover this aspect by describing the aforementioned subjects and apply them into Malmo Project framework, specifically designed for the virtual game Minecraft. We will attempt to use aspects of reinforcement learning to teach an agent specific behavior in order to test the functionality of one of the game's feature.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Master Thesis, March 2018, Utrecht, Netherlands
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

The paper proceeds as follows. In Section 2 we present our research goal. In section 3 we perform a case study, where we analyze the related work trying to answer RQ1 and RQ3. In section 4 we present our research approach explaining details about our algorithm. Experiments and their results are analyzed in Section 5. In section 6 we briefly discuss about the limitations and future work and we conclude the study in section 7.

2 RESEARCH OBJECTIVE

2.1 Problem

Current game testing practices are labor intensive and become tedious and monotonous with the passage of time, since the testers are required to play a game (or even the same scenarios) several times to test the various versions. Manual test case generation is a monotonous and error prone task, is not systematic and hence is not scalable. This becomes a major challenge when changes appear frequently. In addition recently AI techniques are mostly focusing on building smarter agents in video games. Although, they succeed in older and simpler games like 2D platformers, AI has only recently began to advance in more complex, open world games. Currently there are not many game industries that use the combination of AI and automated testing techniques to test their games.

2.2 Research questions

Our main research question are as follows:

RQ1. *Is automated testing being used in 3D virtual worlds and how?*
RQ2. *Can reinforcement learning contribute in building an AI agent that autonomously play and test its features?*

The first question aims to investigate whether automation testing techniques are being currently used in vast open world virtual games. Is automated testing being used in the game industry? How efficiently is it being used and what does it offer to the testing procedure? Going deeper into the literature in section 2, we will discuss several testing techniques that facilitate the automation process and the possible advantages and limitations.

The second question is concerned with finding a suitable AI technique that can be used to train an agent to learn to play a video game by itself. Given a set of rules, the agent should grow a behavior and learn to perform specific tasks related to functional testing. Is reinforcement learning the optimal technique? Will the agent be able to sufficiently perform testing through this process?

Another minor research question that occurs from **RQ1.** is the following:

RQ3. *How is testing coverage applicable in vast open world environments?*

Since the complexity of an open world game environment is enormous, especially in AAA games, is coverage of automated functional testing applicable? What techniques can be used? How easy is it to test every feature of a game and how can we achieve high coverage in such environments with a low time cost?

In the next sections we will try to answer the above questions.

3 RELATED WORK

Software testing is an important technique for analyzing a software product to detect differences between existing and required conditions (bugs) and evaluate the quality of the software product. Software testing is critical task in the software development process and considerably imposes cost and time restrictions on the development process. Therefore, automating the testing process can significantly increase testing process performance. But before we dig into that, we should first discuss about the two basic classes of software testing: black box testing and white box testing [31] . These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

3.1 Black vs White box testing

Black box testing[35], also referred to as functional testing, is a software testing method in which the internal structure of the item being tested is not known to the tester. The method focuses solely on the outputs generated in response to selected inputs and execution conditions. This method is named so because the software, in the eyes of the tester, is like a black box , inside which one cannot see. With black box testing, the software tester does not have access to the source code itself. He only knows the information to input into the black box and what to expect as an outcome based on the requirements knowledge. For example, a tester, without knowledge of the internal structures of a website, tests the web pages by using a browser, providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome. In black box testing, test cases can be designed as soon as the specifications of the software product are complete, although without clear specifications, test cases might be difficult to design.

White box testing[35], also referred to as structural testing, is a software testing method focusing on the internal mechanism of a software product. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Unlike black box testing, the tester, possibly the developer of the code, has access to the software code and writes test cases by executing methods with certain parameters. This testing method is more thorough, with the possibility of covering most paths but it can also proven to be more complex.

In overall, neither structural testing nor functional testing is by itself good enough to detect most of the faults. Therefore, it is imperative to combine both of the techniques to achieve a better testing result.

3.2 Levels of testing

Different levels of testing are used in the testing process and each level of testing aims to test different units of the system. A software system goes through four stages of testing before it is actually deployed. These four stages are known as unit, integration, system, and acceptance level testing.

In unit testing, white box testing techniques are commonly used to test individual software units such as procedures, functions, classes etc.within the software itself. This is the first level of testing and greatly improves the reliability of the code [22].

The fact that units might work well individually doesn't necessarily mean that they work when combined and that's where the

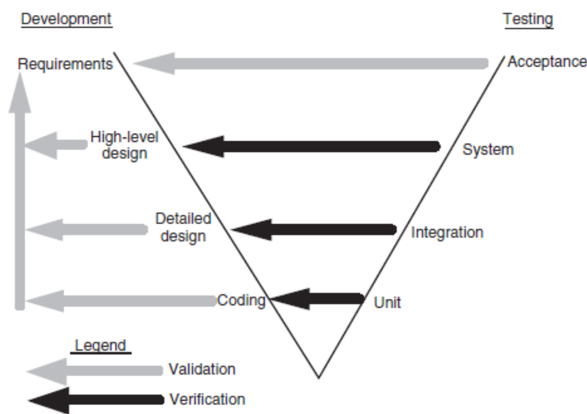

Figure 1: Levels of testing.

second testing level, integration testing, comes into play. Integration testing, is used to verify that units work together when they are integrated into a larger code base.

System testing, is used on a fully integrated software system to evaluate the design and ensure the functionality of the system according to the specified requirements. System-level testing includes a wide spectrum of testing, such as functionality, stress, performance, usability, security testing etc. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

Acceptance testing, uses black box techniques to determine whether the software system meets the acceptance criteria. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing.

3.3 Regression testing

Regression testing is a 'sub-level' of testing, which is performed whenever a component of the system is modified to verify that the modification has not introduced any new unintended faults and that the system still complies to the specified requirements. Regression testing can be considered as a testing technique, rather than level, as long as it is performed at any of the previously discussed levels. In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software.

3.4 Combinatorial testing

Another noteworthy testing method is the combinatorial testing (CT) method [32] which designs tests for a system under test (SUT) by combining input parameters. For each parameter of the system, a value is chosen. This collection of parameter values is called test case. The set of all test cases constitutes the test suite for the SUT. Instead of testing all possible combinations of values, a subset of combinations is generated to satisfy some well-defined combination strategies. Another advantage of CT is that it can detect failures triggered by the interactions among parameters in SUT [24]. Combinatorial design can dramatically reduce the number of combinations to be covered but remains very effective in terms of fault detection.

3.5 Coverage

In traditional software testing, test coverage is a measure of the proportion of a program exercised during testing. It provides us with an objective score (percentage) of the amount of testing performed by a set of test. However, coverage doesn't help in finding errors and bugs in the system under test. Developers and testers try to achieve 100% testing coverage, but that doesn't mean that all possible bugs in a program have been found. Especially, in video game industry this is not a trivial task. How can we apply testing coverage into modern virtual worlds? Before we dig into that we first present a useful software testing technique.

Equivalence partitioning (EP) is a very widely used black-box technique that can be applied at any level of testing and is used to decrease the number of possible test cases that are required to test a system . The idea behind this technique is to divide (partition) a set of test conditions into groups or sets that can be considered the same.

Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification and will be treated similarly by the component. Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition. In simpler words, in equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition. EP is a great technique for reducing the number of test cases required to test a system, while allowing for a greater testing coverage of the system.

Now let's try to consider what happens in video games. Games are usually simulations and rely on massive amounts of shared state which can't be tested in isolation. When a game is typically very complex and has many states, it is just not feasible to cover all the different states the game can be in. The variables that affect the game state are theoretically infinite thus it is not possible to test every combination to achieve 100% test coverage.

When we narrow it down to our case, let's imagine that an agents has a set of items in the inventory and wants to combine them to create new items. The input space (items in inventory) might be big but we can only have 2 output options, either we combined successfully or not. Now this example might seem trivial, but what happens if the output is not that trivial? If the output space is also enormous we simply don't have the time and resources to test every combination. The complexity of such process is pretty high as we can't combine every possible input and output to achieve maximum coverage. A good solution would be to use EP to narrow down the

number of test cases required for the system under test. In any case, we should also use state of the art AI techniques to reduce the time cost of the test suite.

To conclude we can only assume that test coverage of interactions of an agent within the game universe will be higher if human testers are being used. To automate this process we need high level calibration and accuracy on modeling test suites that will provide us with high coverage.

3.6 Game Testing Examples

Game testing is a subset of game development, a software testing process for quality control of video games. Its primary goal is to detect and document any possible software defects (bugs). Game testing will test for issues such as functionality, performance, compatibility, consistency, completeness, and will reveal potential programming bugs.

However game testing is not a trivial task. One of the most complex aspects of game testing can be the testing of the actual world or level, especially if it is a vast, sprawling, 3D world, such as for modern MMOs and RPGs. Some parts of this can be automated, such as having bots move randomly through the game world to see if they get stuck or find other problems with the world. As the complexity of the task grows, it becomes more and more important to find ways to reduce the complexity with the help of tools[].

There have been several research projects in this area which influenced our work. W.K. Chen et al. [36] propose a method that automates the testing process of a game. They designed a HTML5 game framework (H5GF) with a testing layer that can execute test scripts to perform user events and assert the correctness of the game features. With this framework, a tester can create a testing script by either write it directly in JavaScript or by playing the game and capture the gameplay actions and translate them into a testing script. Their main contribution to the automation testing problem is that their framework is game-independent and functional testing can be achieved as long as the game is developed using this framework. Writing a test script directly might be a hefty job, thus the authors provide the ability to capture a test script during gameplay. When an event is triggered, the information of the event is stored and translated into a corresponding test script. However the tester should manually add assertions into the script to ensure the correctness of the game states. In case of any change in the code of the game, all test cases have to be re-recorded and evaluated manually, which is neither efficient nor scalable, although the authors argue that this method is considered to be much faster than writing the test script directly. In addition, due to the dynamic nature of games, timing plays an important role in the success of the framework. If timing is not exact as in the recorded test scripts mainly due to performance issues, the framework might not reproduce the same results, thus adaptation in real time is required. The maintenance effort of such recordings is high and prone to playback errors. Nevertheless, their insight on capturing looks really promising and should definitely be investigated more in the future.

Nantes et al. [23] use a semi automatic testing process with pattern recognition.They propose a general framework for addressing the automation of the game testing problem by using Artificial Intelligence (AI) and Computer Vision (CV). They present an approach for dealing with the automatic detection of environment anomalies making the video game testing process semi automatic. Their approach covers not only the functional aspect alone, but other aspects such as gameplay mechanics and functionality (Entertainment Inspection) or visual anomalies (Integrity Inspection). They aim to give the agent vision-debugger features and combine techniques from AI and Computer Vision along with low level data disclosed by the video game. They call this approach the Sub-Representational approach. They analyze the same images perceived by a human player from the game, in order to extract characteristic information, detect certain events and evaluate the integrity of the virtual environment. They manage to achieve that by using noiseless data directly from the GPU pipeline and the drivers instead of using real sensors. Specifically, they access the graphic pipeline by modifying the graphic drivers in such way to be able to store the traffic data that pass through. Moreover, in order for the agent to be able to use this information effectively, they use a characterization for each bug so it can be recognized in an unambiguous manner. Sometimes, for bugs particularly complex to characterize, they pre-process the image in order to highlight or isolate characteristic objects or features in such a way as to facilitate the detection process, hence making it more robust. This is achieved by utilising the Shadow Map technique, used for casting shadows on arbitrary meshes in order to detect visual anomalies during the pre-processing stage. In general their approach looks intuitive enough but it differs a lot from our work as we are not dealing with computer vision aspects.

Another interesting work is the one of S. Iftikar et al. [14]. In their work they propose a model based testing approach for automated black box, system-level, functional testing on platform games. The proposed model allows for automated test case generation, automated test execution, and automated test oracle (test verdict) generation. The authors use domain modeling for representing the game structure and UML state machines for behavioral modeling. More specifically, they construct a profile in UML that includes both conceptual and behavior details of a game, such as actions, collectibles, user/system generated events etc. Then, they use the above model to automate their testing strategy. Their approach is divided into three basic steps. In the first step, they use the state machine of an AI agent in Super Mario Brothers game and the game's state machine to generate test cases. With the use of record and replay tool to record the initial set of steps for the agent and the N+ testing strategy, an approach used for generating test cases based on state machines[4], they manage to produce a transition tree that contains end-to-end paths from the state machine. From the state machine, they generate test paths using round trip and sneak path strategies to navigate the agent through the environment. The second step consist of creating test oracles that are triggered as a result of the user-generated events. In this step a testing interface is being used to provide the details of the internal game states during testing. The last step is the test execution. During test execution, if the system fails to generate a system-generated event corresponding to a user-generated event or the testing interface fails to verify a state change corresponding to a transition or the application crashes, the application will be considered to have failed the testing. The tester is able to define a time period for individual test case execution, during which the tester may select various random test sequences and execute them. The above framework was tested in the very famous

Super Mario Brothers game and managed to identify bugs during game testing. Overall, their profiling and modeling methodology combined with the test case generation and execution looks very promising and applicable to many platform games. Unfortunately in our case, it will not be sufficient enough as our game is an open world complex environment that require a different approach.

Our major inspiration was Udagawa et al. [31] project, in which the authors use Malmo Project framework to kill as many enemy zombies and survive their attacks for as long as possible using reinforcement learning. Their objective is to teach the agent what policies to execute based solely on the pixels of the gameplay screen. Rather than giving the agent information about the environment or any guidance on killing zombies, it has to learn optimal actions based on its past experience. They achieve this by combining Q-Learning and neural networks to teach evaluate the input image that they receive from the observations of the agent, and optimize the Q-values, so that the agent will learn which action is most effective in killing enemy mobs. Using Q-learning and experience replay methods, the agent showed a significant improvement in killing enemy zombies and surviving for a long time.

We tried to create our agent in a similar approach utilizing some of their ideas into our algorithm. Later on, we discuss the implementation and how this process works for us.

In addition Mnih et al. [21] present a deep reinforcement learning model that successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network that overcomes the challenges of RL, to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the Q-learning algorithm [34], with stochastic gradient descent to update the weights. They use an experience replay mechanism which stores the agent's experiences at each timestep into a replay memory pool. Then they randomly sample the pool and apply the Q-learning updates, this manages to smooth the training distribution over past behaviors. The model is put to test in six Atari 2600 games and compare the results with those of human play managing to achieve better results that the human in half of them.

4 RESEARCH APPROACH

4.1 The Framework

For the purposes of our study we chose to implement our research ideas into Minecraft, a widely popular sandbox video game, released in 2011. The main aspect of the game is the ability of the player to create and build construction out of textured cubes in a 3D procedurally generated world. Other activities in the game include exploration, resource gathering, crafting, and combat. The game has no specific goals for the player to accomplish, allowing players a great amount of freedom in choosing how to play the game. Because Minecraft offers so much creative freedom to players along with it hugely diverse challenges and creation opportunities, it appears to be a great proving ground for an AI-controlled character to roam, learn, and evolve. Minecraft can be viewed as an approximation of the natural world, thus our goal is to implement our agent as if it is learning from and interacting with the world in a human-like behavior.

For this purpose we used the framework called Project Malmo, which is an open source experimentation platform, designed by Microsoft Research, specifically to leverage Minecraft as means to improve the AI problem solving. It is built on top of Minecraft and provides an interface for researchers to experiment with different approaches in artificial intelligence.

Through the Malmo Project framework, a researcher can experiment with AI bots, using deep reinforcement learning and other agent-based AI techniques, create environments and combine different states, actions, and rewards to simulate tasks for which the agent trains with reinforcement learning to teach the AI agent to learn to play the game by itself. More specifically, AI agents are allowed to complete extensive scenarios and error explorations with a given task and will receive a reward for completing it successfully.

Malmo treats the agent as a separate entity from the virtual world meaning that it can only get information about the world through observations of different states and rewards in real time. Furthermore, the fact that it is build upon a vast open virtual world where the agent is trained on its first person visual input contrasts with other related work like Atari games tested by DeepMind [21] or 2D flash games tested in OpenAI, where the games provide a bird's-eye view. These factors make Malmo a difficult but interesting platform to build upon and experiment with.

We chose this framework for two reasons. Firstly, as we see in the literature review section (Appendix), RL is believed to be the most promising approach to reach our goal. Secondly, the framework itself pushes us towards that direction by providing us the necessary tools to apply RL techniques with ease.

In more technical details, the framework uses a high level concept called MissionSpec to specify the mission for an agent to solve. This may include information about the map, consumables and items, rewards etc. All the above information can be specified in XML by the user to ensure the compatibility across the agents. The AgentHost instantiates missions using the MissionSpec to bind it with the agent. During the mission the agents interact with the AgentHost to observe the WorldState (time stamp for mission, observations, rewards) and execute actions. Such actions might differ from collecting an item, to placing blocks or even attacking enemy monsters. All log information is stored into MissionRecord.

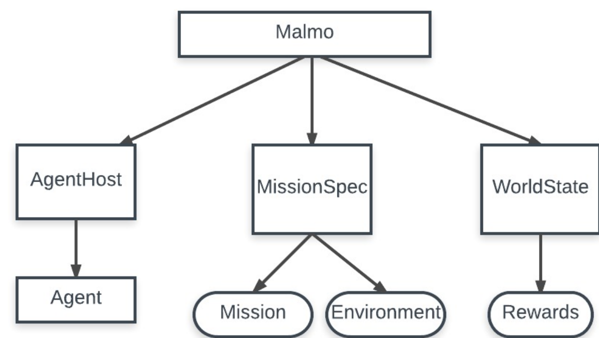

Figure 2: Malmo project class hierarchy.

The framework supports many programming languages such as Java, C++, CSharp and Python. We chose to implement our approach using the Python language as it is heavily used by fellow researchers and provides more capabilities to the framework.

4.2 The Algorithm

In this section we provide a rough representation of the algorithmic steps of our approach and we further analyze our implementation process.

```
while repetition < max repetitions do
    setup agent and start mission;
    while current reward < reward goal do
        roam and perform actions;
        perform obstacle avoidance;
        get new observations;
        if receive new observation then
            get world state;
            calculate current reward;
            if current reward > reward goal then
                perform elimination;
                invoke functionality under test;
                update Reward Matrix;
            end
        end
    end
    check ending condition;
    check success of functionality under test;
    reset world information;
end
```

Algorithm 1: Rough pseudo-code representation of our algorithm

The algorithm represents a generic approach on automating the testing process of a test case according to the tools and capabilities of the framework we used. This means with slight changes to the algorithm we can invoke different functionalities to test.

The roaming is not a trivial job in this framework. There exist a lot of different ways to make the agent move inside the world. In our case the agent moves freely in the environment being able to jump over obstacles and climb big structures of blocks. That way, the agent is tested in a realistic 3D world environment. According to the setup of the world environment, we should change the path planning of the agent to avoid possible deadlocks in it's path such as being trapped into corners or inside structures where the entrance/exit is not easily accessible. To make the agent even smarter we didn't just make it move randomly but we tried to dictate its movement according to certain rules/conditions. In the next section we explain exactly how the movement of the agent is implemented.

There is a finite number of actions that an agent can do in the framework and this is just a subset of all the possible actions that Minecraft supports. Those actions namely are: move, strafe, pitch, turn, jump, attack, use, discard.

When the mission starts the agent will receive observations through the framework every game tick. Those observations contain useful information about the the world state and will be used by the agent to dictate its behavior. Whenever it finds that there is a new observation, the framework provides the functionality to check the current state of the agent and calculate the cumulative reward. Observations may vary from receiving feedback about entities that are close to the agent to collecting items or getting rewards from completing specific set of actions.

An elimination process is used to rule out possible actions or attributes that might not be necessary for the functionality under test. In regards of the use of delta debugging, we try to test specific sets of attributes and rule out those that don't affect our result. This way we systematically check the functionality of the feature in different states of gameplay. This process if very generic and can be modified and applied according to our needs. In the next section we discuss this process in our specific scenario.

We should note that our algorithm does not use any traditional reinforcement learning algorithm such as Q-learning. In reinforcement learning, an agent interacts with its environment by perceiving its state and selecting an appropriate action, either using a policy or by randomly selecting an action from a set of possible actions. The agent receives feedback in terms of rewards, which rate the performance of its previous action.

In our case we tried to differentiate slightly. Although the concepts of observations, actions and rewards still exist, we manipulate them to meet our demands. Our algorithm doesn't translate the information of a reward as an indication that an action was "good" or "bad". Here a reward represents the desire of the agent to do a certain action. Typically, the Reward Matrix should be adjusted according to the rewards of completing specific actions. In our case the Reward Matrix serves as a knowledge base for the agent providing useful information about what consists of a "good behavior" for our functionality under test.

In section 6 we discuss why did we take this decision and what were the limitation of such an approach.

4.3 Our Scenario

In our work, we will use the algorithm to test the functionality of the "craft" command. The agent will pick up some of the ingredients that are scattered around the world and try to combine them to craft a specific item. Through an elimination process, our algorithm should give an output of the recipe. The recipe consists of a set of items that are required in order to craft an item. It's an indication of what items should be in the inventory for the crafting to be successful. The agent has to collect them and try to combine them. For the ease of our work, we do not experiment with complex recipes that require more than one instances of an item. We assume that we need only one instance of an item in our inventory.

The basic information about the mission is given in an XML format at the beginning of the test case. Those information concern the creation of the world, the agent basic information such as his spawn position and the observations that the framework will provide us with. *Figure 3* presents an example of a mission.

The user/tester is able to manipulate certain parameters on our script, referred as independent variables, to control the test case. The agent considers the world as a black box, he has no knowledge of those variables as well as any other information about the world

```
def GetMissionXML(summary, itemDrawingXML):
    ''' Build an XML mission string that uses the RewardForCollectingItem mission handler.'''
    xml = '''<?xml version="1.0" encoding="UTF-8" ?>
    <Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <About>
            <Summary>''' + summary + '''</Summary>
        </About>
        <ServerSection>
            <ServerHandlers>
                <FlatWorldGenerator generatorString="3;7,220*1,5*3,2;3;,biome_1" forceReset="true"  />
                <DrawingDecorator>
                    <DrawCuboid x1="-50" y1="226" z1="-50" x2="50" y2="226" z2="50" type="carpet" colour="RED" face="UP"/>
                    ''' + itemDrawingXML + '''
                </DrawingDecorator>
                <ServerQuitFromTimeUp timeLimitMs="200000"/>
                <ServerQuitWhenAnyAgentFinishes />
            </ServerHandlers>
        </ServerSection>

        <AgentSection mode="Survival">
            <Name>Eat Agent</Name>
            <AgentStart>
                <Placement x="0.5" y="227.0" z="0.5"/>
                <Inventory>
                </Inventory>
            </AgentStart>
            <AgentHandlers>
                <VideoProducer>
                    <Width>800</Width>
                    <Height>600</Height>
                </VideoProducer>
                <RewardForCollectingItem>
                    <Item reward="1000" type="pumpkin_pie"/>
                    ''' + GetReward + '''
                </RewardForCollectingItem>
                <ContinuousMovementCommands turnSpeedDegs="240"/>
                <SimpleCraftCommands/>
                <InventoryCommands/>
                <MissionQuitCommands/>
                <ObservationFromNearbyEntities>
                    <Range name="entities" xrange="5" yrange="40" zrange="5"/>
                </ObservationFromNearbyEntities>
                <ObservationFromFullInventory/>
                <ObservationFromSubgoalPositionList>''' + getSubgoalPositions(positions) + '''
                </ObservationFromSubgoalPositionList>

            </AgentHandlers>
        </AgentSection>
    </Mission>'''
    return xml
```

Figure 3: An example of the XML mission

and should be able to find out which exact ingredients are needed for the recipe due to our AI algorithm. Those parameters are:

(1) The area *A* of the world where all the items will be generated
(2) The number of the items *Ni* that can be generated
(3) The number of the types of items *Nt* that will be spawned
(4) The desire threshold *Dth*
(5) The reward goal *RwG*
(6) The Policies *Pi* needed for updating the Reward Matrix

We chose to implement our scenario into a 50x50 open world environment with various obstacles and structures of certain height such as hills or ladders. However,since navigation wasn't as an important aspect as the functional testing and AI techniques in our experiment, the AI algorithm of the path planning is pretty simplistic, providing the agent to jump over obstacles and block structures. A far more complex world would require heavy navigation calibration which falls out of the scope of our research.

Each item is randomly generated into the world and scattered around across the area A, a parameter controlled by the user prior to the experiment. Each type of item eg. melons, apples, eggs etc., is given a randomly generated float number between 0 and 1 as a reward. This number also reflects to the Reward Matrix and can be translated as a desire of the agent to pick up the specific item. The higher the reward value is, the higher the probability of being the item being picked up by our agent. The Reward Matrix is basically a python dictionary that registers the name of every type of item available in our test case and its reward value.

The desire threshold is a float number, given by the human tester, that represents the threshold above which the AI agent will try to find a type of item to collect.

The reward goal is a constant float number, given by the tester which will serve as an ending condition for every iteration. When the current reward sum exceeds the reward goal, the agent will stop moving and collecting items and will attempt to craft.

We used two different policies to update our Reward Matrix. In the first policy *P1*, after every repetition of the mission (see algorithm), we set the new reward value of every type of item to a random float number between its previous value and 1. That way, as repetitions go by, the reward value of the item increases and will be forced to exceed the desire threshold and ensure that the item will eventually be picked up. On the contrast, the second policy that we used *P2*, after every repetition, randomly generates a new reward value for the item.

The agent can perform six actions: move forward, backwards, turn left, right and pickup or discard items and craft. We chose to make the agent move and turn with a constant speed at each game tick while roaming around the virtual world.

While the mission is running, our agent receives information about the blocks that exist in the world in a 3x3x2 grid away from his current position. For example, he gets information about the blocks that are directly in front of him on his current height and the one above him. That way the agent knows whether he can actually move forwards or jump over an obstacle. This information comes in a form of a 2D array that contains the names of the adjacent blocks in the specified height next to the agent. We translate this array in a way to understand whether the agent is able to jump over a block and the direction it needs to follow. In the extreme conditions that an agent cannot get past or over certain obstacles (possibly a row of blocks with height more than 2), after a few game ticks our algorithm can determine that the agent is stuck in a deadlock position. We then make the agent either turn left or right so he has the opportunity to find a way to avoid the obstacle by recalculating his new route.

In every repetition, the agent starts from the center of our work space and moves around the world to collect items. Its movement is predetermined offline at the beginning of the running test script. Specifically, before each repetition our algorithm saves in a list of positions, the coordinates of every item that has a desire above the given threshold *Dth*. The agent will visit those positions and will collect every item that it finds on its path. That way we ensure that that the agent will have the opportunity to collect every type of item, even the ones with low desire, which might prove to be an ingredient of the recipe.

Our algorithm calculates the reward value of each item when it receives a new observation and adds it to the sum of the current reward. The following formula represents the calculation of the current reward value:

$$CR = Pcr +_{i \in Inv} R_i * N_i \tag{1}$$

'CR' represents the current cumulative reward, 'Pcr' the previous current reward ,'R' is the reward of an item and 'Ni' the number of instances of an item that have been collected since last observation.

Once the total reward exceeds the reward goal the agent will stop moving around and will attempt to craft the desired item.

The agent so far has no knowledge of the recipe and the items that are needed to craft the item. In order for the agent to find out which one of the ingredients that it gathered are part of the recipe, we implemented a solution based on the idea of delta debugging. Considering the current state of the agent's inventory as our unit under test, we will try to narrow down the set of items that are

needed until a minimal set remains that will still validate the crafting command. To achieve this,the agent will toss the first item on its inventory that hasn't been tested before and try to craft.

If the desired item is crafted, that means that the item that it discarded is not needed for the recipe thus it is irrelevant in our world. We then update the reward matrix, changing the value of that item to 0. This way the item will never be picked up again as it is not needed.

On the other hand, if the desired item is not crafted, that means that at the current moment, the agent does not have all the recipe ingredients on its inventory. In this case the agent either didn't pick up all the items or the item that it tossed is required in the recipe. We again update the reward matrix, changing the value of the discarded item to 1. This means that in every other repetition this item is of high desire as it it possibly of the ingredients of the recipe.

We call this the elimination process as in every repetition the agent will learn a new information about the ingredients of the recipe by eliminating the possible candidates one by one.

This process repeats for several iterations until we reach a point where the Reward Matrix contains only values of 0 and 1. That means that we eliminated all the possible candidates for the recipe. Then we perform a check to investigate if the agent came up with the correct recipe for the crafting.The results are shown in the next section.

In table 1 we show an example of the reward matrix in different states of the testing phase. This table is just a representative example we use for ease of understanding.

Item	starting Reward	ending Reward
pumpkin	0.758	1
beef	0.803	0.989
sugar	0.148	1
egg	0.657	1
carrot	0.235	0.870

Table 1: Example of Reward Matrix in different phases. The rewards value of every item at the beginning of the test and the resulting Reward Matrix. From the ingredients with reward value 1 we can derive the recipe.

5 EXPERIMENTS

To answer the research questions and evaluate our research approach we conducted several experiments, which we will refer to as test cases. Because the total amount of test cases is enormous due to the number of variables that affect the algorithm, we decided to cover a few important test cases that we believe will help us answer our research questions.

The goal of the experiments is to test the functionality of the "craft" command. The first set of experiments serve as a training session where the agent will pick up some of the ingredients that are scattered around the world and try to combine them to craft a specific item. Through an elimination process, our algorithm should give an output of a set of items that represent the recipe. In the end the ingredients of the recipe are stored in a json file. In the second set of experiments we will use a more simplistic version of our algorithm. In these experiments the agent knows the recipe provided by the training session and will try to determine if it is actually correct in various scenarios. We will attempt to measure the accuracy of our algorithm to determine the success or failure of crafting in extreme situations.

Figure 4: Different states of gameplay. The picture on the left shows a state during the mission. The picture on the right shows a state where our agent successfully crafted the desired item (in this case a pumpkin pie).

As a baseline configuration for all experiments, we chose to keep some of the independent variables constant in all test cases. For instance, in every test case we generate all the items in the same 50x50 blocks distribution area. We also keep the desire threshold to a fixed value of 0.75. Finally the number of instances of an item for every type stays the same in all test cases. For example, in a configuration of 100 items with 10 types of unique items, we will spawn 10 instances of every type of item.

We chose three metrics to measure the agent performance over the experiment:

(1) The average time needed to come up with the recipe
(2) The amount of times that the agent successfully found the exact recipe

For the first set of experiments we tested our algorithm in a configuration of 10 types of available items in the world space seeking for crafting of an item that needs 3 different types of items. We only changed the total amount of items that are spawned in the world starting from 100 and increasing the number by 50 until we reach 300 as shown in 2. Each configuration run 4 times in which we measured the time needed for every iteration until the agent comes up with a recipe. We then measured the average time needed for the run for each total number of items in the world.

Independent variable	Value
Types of items	10
Number of items	100,150,...,300
Policy	P1,P2
Number of items in Recipe	3

Table 2: Experiment 1 configuration.

We found that the algorithm performed better when a large amount of items are spawned in the world. With 300 possible items

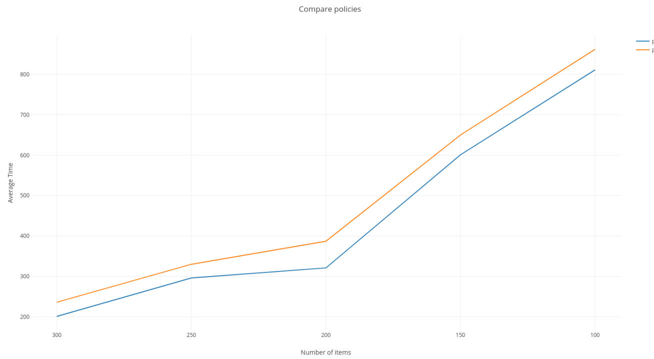

Figure 5: Comparison of policies. The blue line represents P1 and the orange P2. We see that P1 performs much better in average time than P2. The timings are in seconds.

the algorithm needed an average of **3 minutes and 35 sec**. While we decrease the number of possible items the algorithm takes more time to come up with a recipe. For example with 200 available items the average time needed was **5 minutes and 35 sec**, while the time dramatically increased with 100 items with an average of **13 minutes and 51 sec**. Another interesting result was the time the agent needed to reach the reward goal and start the elimination process. We found out that in 300 items the time span ranged from **14 to 33 seconds** while on 100 items it took **50 to 122 seconds** for every iteration.

With the above configuration the agent was able to identify the exact recipe 80% of the runs. In the other 20% the agent only added one more item in the recipe.

The previous test case used the policy *P1*. We then decided to compare *P1* with *P2* in the above configuration. The results are shown in *Figure 5*.

We can clearly see that *P1* performs better in time than *P2*. That is because in policy *P1* the reward value of every type of item is increased over time meaning that the probability that the reward will exceed the reward threshold will be higher, therefore the item will be on the list of items that the agent will definitely try to pick up during the run. On the contrast with policy *P2*, we noticed that it might take more time for the reward to be adjusted above the threshold, thus it is more likely that the agent will not try to find it and collect it. In fact the less the available items in the world the less efficient is our policy with it poorly scoring an average of roughly **14 minutes** for only 100 items.

We can acknowledge that this is a very time consuming process to test only the crafting feature. In the next section we discuss a few ideas for future work, to find a more suitable policy for updating the reward matrix that might lead in improving the total running time.

In our next experiment we investigated how the number of types of available items in the world, affect the average running time. In this configuration we used 100 and 300 items with 5,7 and 10 available types of items. The results can be shown in *Figure 6*

We can clearly see that with less available types and given a large amount of items spawned in the area, a recipe that requires 3

Independent variable	Value
Types of items	5,7,10
Number of items	100,300
Policy	P1
Number of items in Recipe	3

Table 3: Experiment 2 configuration.

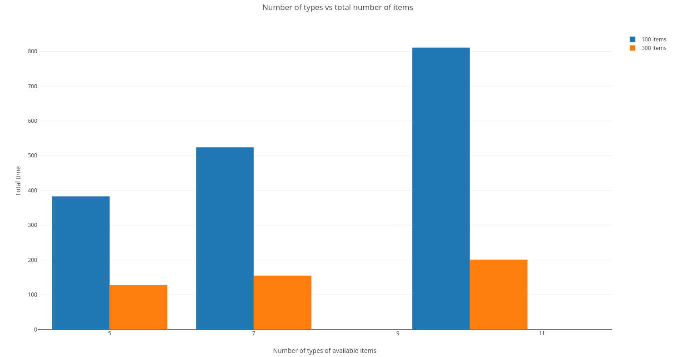

Figure 6: Evaluation of the algorithm with different number of types of available items in the world. The blue bar represents the average time with 100 items and the orange with 300 items

type of items performs much faster. This also reflect to the number of iterations that were needed. The more types of items available in the world the more iterations are needed due to our elimination process. The algorithm predicts the correct recipe in 90% of the times with 300 items and 75% with 100 items.

In our final experiment we experimented with a different recipe scheme. This time we explore the complexity of the algorithm when the number of ingredients of the recipe changes. We tried 3 different recipes: one that needed 2 items, one with 3 (which is also the case in our previous experiments) and one with 4.

Independent variable	Value
Types of items	7
Number of items	200
Policy	P1
Number of items in Recipe	2,3,4

Table 4: Experiment 3 configuration.

The results indicate that the complexity of the recipe doesn't affect directly the average running time and iterations are needed to complete our task, due to the nature of the elimination process that we use. For the same reason the algorithm finds the exact recipe in an average of 75% of the times. Our only notice is that, the lower the number of items needed in a recipe the more likely the agent will add more unnecessary items in the resulting recipe and result in a lower success score. That finding means that the complexity of the recipe isn't a factor that will affect greatly the performance of the algorithm, which is clearly an advantage of our method.

In the second set of experiments, we will attempt to measure the performance of our algorithm while injecting an artificial error to our cases. This error is represented by the situation where the agent is not able to pick up a specific type of item.

We consider a binary classifier that classifies each input pattern in a data set into two classes, either positive (P') or negative (N'), while the ground truth is either positive (P) or negative (N). The performance of the classifier can be represented in terms of these four possible classification results:

(1) True positive (TP): the result is positive (P') while the ground truth is also positive (P)
(2) False positive (FP): the result is positive (P') but the ground truth is negative (N)
(3) True negative (TN): the result is negative (N') while the ground truth is also negative (N)
(4) False negative (FN): the result is negative (N') but the ground truth is positive (P)

In our scenarios the ground truth represent our world environment and its accessibility. More specifically, a positive ground truth represent a world where the agent can climb every obstacle and is able to reach every available item from at least one path, while a negative ground truth represent a world where an inaccessible area exist, so the agent will never have the chance to collect an item inside this area. Accordingly a positive result indicates the success of the crafting command and the negative result the failure of crafting.

Now the agent knows the recipe provided by the previous training session. In these experiments our algorithm is much simpler. The agent will have to pick up some of the items that are scattered in the world in the same fashion as in the training session. For the simplicity of our testing, we will only spawn the type of items that belong to the recipe and the agent will have to try to collect all of them. Once our algorithm reaches an ending condition that we set, the agent will stop moving and simply try to craft the desired items. Our algorithm is able to determine the success of the crafting command and make an assessment of what is wrong if the crafting fails. Those ending conditions are heuristics that will determine the search of our agent. We used two different conditions/heuristics and those are a reward goal and distance travelled. We should note that, because the game will never fail to craft an item as long as the agent has all the necessary items, we had to inject some kind of artificial error to our test oracles to simulating error prone situations. The world environment that we used is similar to the one of the previous session in terms of size and the structures/obstacles in it but with small variations for every different test oracle.

In our based our experiments in crafting an item that requires 3 types of items and run 20 single simulations for each test oracle to measure the performance of the algorithm.

In our first test oracle we will test the performance of our algorithm using the heuristic of the reward goal. In this situation, every type of item spawned in the world receives a random reward number between 0 and 1. When the agent collects an items we calculate the accumulative reward. Once the accumulative reward exceeds the reward goal that we have set, the agent will stop moving and try to craft. We experimented with the variables that affect our experiment namely the number of instance of items in our world and the reward goal. We found out that the lower the reward goal the less chances the agent has to collect the necessary items. For example with the reward goal set to 1 the algorithm performs poorly in positive ground truth, with only 30% of achieving a true positive (TP) while if we increase the the reward goal to 2 the TP rate increases to 80% with only 10 instances of items in the world. In a similar fashion, the more items spawned in the world the better the TP rate is as long as the reward goal is set above 2.

The second test oracle we test the performance of our algorithm using the heuristic of the distance travelled. We measure the in-game distance our agent has travelled and set up a threshold where the agent should stop searching for items. When the agents exceeds this value, it will stop moving and attempt to craft. In this situation, there is no reward value to any of the items as we don't calculate any rewards.

We found out that the greater the distance threshold is the greater the TP rate is with a rate of 95% on a threshold of 5000 meters (in-game metric) and 80% with 3000 meters. In all experiments we only spawned 10 instances of items in the world. While decreasing this threshold, we give less chance to the agent to actually pick up all the necessary items so the false negative (FN) rate increases dramatically.

In our final set of experiment, we created a world with a secluded area where the agent could not enter from any direction. WE then spawned some items in this area and run several simulations. As expected the rate of false positive (FP) is pretty low achieving only a 10%. On 90% of the simulations the algorithm gave a true negative (TN) meaning that the agent did not find all the types of items needed for the recipe and failed to craft.

In all the above experiments, we can classify the reason of failing to craft because of the nature of the existing world. However if we set a default world where we don't know if the agent can actually reach any possible place, we need a more sophisticated AI to determine the reason of failure.

6 LIMITATIONS AND FUTURE WORK

To begin with, we must keep in mind the randomness of several variables and factors in our experiments. For example we cannot dictate the exact distribution of items in a certain area. If all the items are equally distributed in an area, the agent might need more time to pick them up compared to the scenario where all the items an gathered in one corner of the area, because visiting this area will make the agent interact and collect all the items that it finds. This will negatively affect the overall testing time.

Additionally, an important factor that affects our experiment is the path planning. We previously mentioned that the agent will visit every coordinate that contains an item with a reward value above the desired threshold. This coordinates are saved into a list prior to the start of every iteration. The problem that occurs is that the instances of that list are randomly generated because the coordinates of those items are generated as such. This would mean that the agent might move around the area in an non logical or human-like behavior. For example it might need to go from corner A to corner B and then again to corner A to get a desired item, when it could spend more time in corner A to get all the desired items and not having to go back and forth. The framework provides a solution

with observations from nearby entities, which gives information to the agent to steer accordingly and visit the nearby item entity. The limitation that occurred when we tried this much clever path planning approach was that there were times when the agent found more than one instance of a desired item nearby so in real time it couldn't decide which way to steer, leading to unwanted path planning behavior such as turning around its own axis or strafing constantly left to right without making a decision where to go. In future work, we want to experiment with this approach and apply a reinforcement learning algorithm such as Q-learning or an e-greedy policy to prioritize the decision making process of the path planning.

The most important limitation of the framework was that whenever the agent restarted a mission, the entities and objects of the world were not deleted. Instead we noticed that we spawned new random objects while the previous ones still existed in our world. That had a negative impact on our experiments, so we decided to reset the world whenever we restart a mission in every iteration. However, this means that we now have a new random world, different from the previous iteration. This means that if the agent made the exact same actions as the previous iteration the result would be different. If we applied traditional RL in this case then we can't guarantee that there is an optimal value to maximize the cumulative reward. In fact, theoretically, there exist an infinite number of values and set of actions to achieve that and finding only one of them does not guarantee success of the functionality under test. That is why we didn't use any traditional reinforcement learning technique.

Future work could also focus on implementing a better policy for updating the Reward Matrix. We considered using neural networks to batch the algorithm with information from a replay memory scheme and improve the way the rewards are being updated. This way we can gather more information about the reward of every iteration and use that information to drive the agent to specific actions and behavior.

In addition, we would like to further extend the part of the AI path planning to a certain point where the agent performs the best available action to reach a specific position. For example, we would like to dictate a clever behavior to the agent in cases it gets stuck in a corner or performs the same set of actions continuously.

In the current study we do not implement coverage measurement. It is possible to use Malmo to log the agent's actions and the observation that it samples. There are techniques such as in [26, 27] that can analyze execution traces to calculate coverage, but additionally to inspect correctness properties on these traces. Such an approach has the benefit that it does not require white box instrumentation of the System Under Test, and it works off line. We would like to extend our work with this in the future.

In a project called FITTEST [33] researchers have tried to combine a number of automated testing techniques, e.g. model inference, evolutionary-based testing, and combinatorial testing to test Internet application with complex UI. It would be interesting to see how such a combination can be extended with agent based testing.

When a test reveals a bug, we still need to analyze the execution that the test induced to determine the source of this error, and then fix it. This process is called debugging. When an agent is used to automate a test, the generated execution can be very long, which makes debugging hard. There are algorithms that we can use to minimize the execution. For example the work in [9] discusses the combination of the classical delta debugging algorithm and the use of algebraic rules. It would be interesting to see how such an algorithm could be translated to a virtual world setup.

Keeping all that in mind, our algorithm could really expand the scalability and solve higher-dimensional tasks. For example we can experiment with scenarios where the agent has to fight enemy mobs, find shortest path inside a maze or even try to build a complex construction successfully. This will provide a better insight to our research ideas and possibly convert into a fully functional testing framework for solving non trivial tasks in complex virtual worlds.

7 CONCLUSION

Our study manages to utilize the notion of reinforcement learning to teach an AI agent to complete non trivial tasks in settings with high degrees of difficulty and with limited knowledge of the environment or game rules.

Our ultimate goal and the purpose of our study was to test the functionality of a game feature. In this specific example we wanted to test the functionality of the crafting command in Minecraft. We reported that in 100% of the cases the crafting command worked perfectly as long as the agent had all the required ingredients in his inventory. The only challenge that occurs, is to build an AI algorithm that will steer the agent accordingly and dictate its behavior in order to pick up the correct required ingredients.

We conducted a small literature study for answering **RQ1** and **RQ3**. We present related work in the field of automated testing in game industry showing some noteworthy methods that have already been used. We conclude that automated testing is indeed being implemented in gaming industry but a generalized approach still doesn't exist. Most gaming companies still prefer to use human testers to test their games. We believe that in the future, with the right tools, automated testing will be a very efficient and cost-less solution for game testing. In addition, our experiments prove that our method can be used in a 3D virtual world environment where path planning is not trivial.

The experiments we conducted tried to answer **RQ2**. We introduced a method to teach an agent to autonomously play the game using principles from reinforcement learning. In a reinforcement learning problem, we take the view of an agent that tries to maximize the reward that it receives from making decisions. Thus an agent that receives the maximum possible reward can be viewed as performing the best action for a given state. In our case, the best action would represent crafting the item. However, due to technical limitations of the framework, we couldn't get any useful information from that and we decided to approach the solution using agent based knowledge and our version of the elimination process as described above.

Since the complexity of the world is pretty high we were unable to perform regression testing for all our test cases. We tried to cover the most important test suites, that provide us with important information about using automated testing in open virtual worlds in terms of time.

The results indicate that the average running time of completing the test case increases as the the distribution of items in the world space is more sparse. In case of a larger environment, the agent would probably require a lot of time to traverse from one place to another and pickup the required items. Furthermore, the complexity of a given recipe doesn't really affect the time to complete the automated test case. In addition, we don't need to test every feature of the game. If we manage to sample enough test cases that validate the functionality of a feature, we have enough coverage to assume that it will always work. Nevertheless the experiments show a promising results in terms of time and we believe that with a few changes in the core implementation of the algorithm, our work could be greatly improved

Our approach is not implemented ideally to test more complex tasks and environments but we believe that, with a little effort in the future, it can be greatly improved and has the potential to be used as a generic testing framework for vast open virtual worlds.

A APPENDIX

The layout of the appendix proceeds as follows: In section A1 we provide additional information about testing in games. In section A2 we discuss a few common AI approaches in video games. Both of those sections are related to our research and are considered an extension of our case study. Section A3 briefly describes path planning of agents in virtual worlds. In the last section we provide some complementary information of our algorithm.

A.1 Testing

A.1.1 Testing steps. Testing plays an important role in the development process of a video game. A game testing is done at different levels of development process to detect various defects and bugs. This process is not always done by the game programmers. Usually they only test a small part of their code. Game testers posses the role of testing every aspect of a game. The testers first plan the tests, execute them on the code and report the found bugs to the developers. The developers inspect the reported bugs, debug to find the bugs in the code, fix the bugs in question and compile a new build of the game. The cycle continues as the testers plan the tests and execute them for this new build of the game [15].

Schultz et al. [28] show that a basic game testing process consists of the following steps:

1. **Plan and design the test:** In each build the design specification could have changed, the game could support new configurations, old features might have been cut and bugs could have been addressed. Because of this, the planning and design of tests should be revisited in each build. The aim of testing is to make sure no news bugs were introduced when the aforementioned changes were made.
2. **Prepare the test:**The code, tests, test related documents and the test environment should be updated and be aligned with one another. When the development team has marked the bugs as fixed for the build, the QA team can start running the tests.
3. **Perform the test:**Testers run the test suites again in the new build. When a bug is found, it is examined, so sufficient information can be provided to the bug report.
4. **Report the results:**The completed test suites are logged and all defects found are reported.
5. **Repair the bug:**The development team debugs the code to find the bug and repairs it.
6. **Return to step 1 and re-test:**When the bugs are repaired, and any other additional features wanted for the build are done, the new build can be released to the team. This starts the cycle again, with new possible bugs to examine and new test results.

A.1.2 Types of errors in video games. To better understand the types of errors that might occur in video games, we should mention the work of Lewis et al. [17] where the authors present a taxonomy of possible failures in a video game. Their taxonomy aims to categorize bugs during gameplay and support that it could lead to new solutions in solving video game bugs. Some noteworthy elements of this taxonomy are as follows:

Object out of bounds: Object out of bounds is a classification of an object being outside of the world boundaries.This category encompasses many common types of failures, such as escaping a map or falling through the floor.

Invalid value change: Invalid value change is a broad term that describes any game event that changes some form of counter in an unexpected way, such as a bullet that should remove health not doing so or collecting a coin that changes the score by 100 instead of 1.

Artificial stupidity: Artificial stupidity represents bugs that are directly related to an NPC performing certain actions that break the illusion of intelligence. Common examples include characters not responding to being shot at, blocking doorways or walking into walls.

Information: The Invalid information access category encompasses failures that allow the player to gain more information than is expected by the game design. For example,this category includes seeing through walls or gaining complete or false information on a game map that should have a fog of war.

Action: Action represents actions being taken while the game is paused as well as scripts executing when they are not allowed to yet. This might lead to invalid states of gameplay.

All the above type of errors, among others, can potentially lead to gameplay errors. However, those errors are not easily detectable by AI agents during gameplay. Those errors, by their nature, are non trivial challenges that an agent might ignore or miss during automated testing if it is not trained to behave like a human manual tester. Thus new technologies and solutions are required to address these issues.

A.2 AI

This section will focus on three of the most popular and commonly used AI techniques used in video games: Neural Networks, Reinforcement Learning and Evolutionary Machine Learning.

A.2.1 Neural Networks. Neural Networks [6] are statistical models that are capable of modeling and processing nonlinear relationships between inputs and outputs in parallel. Neural Networks are characterized by containing adaptive weights along paths between

neurons that can be tuned by a learning algorithm that learns from observed data in order to improve the model.

Although neural networks have been predominantly used for game agent representation, they provide efficient operation when trained on sufficient data. Training neural networks with traditional learning algorithms has proven to be inefficient and requires the generation of datasets containing examples to cover all possible situations that may be encountered by the game agent in order to obtain fully robust game agent behaviours. The use of neural networks appears to be well suited when applied to player modelling, though such an approach requires a comprehensive dataset for training and validation. Traditional neural network training algorithms have been shown to be unsuitable for real-time use and are restricted to use as an offline learning mechanism.

To get a better understanding of the applications of Neural Networks in video games, we discuss the work of Geisler [12] "Integrated machine learning for behavior modeling in video games". In this paper, the author show that a subset of AI behaviors can be learned effectively by player modeling using the machine learning technique of neural network classifiers trained with boosting and bagging. The model is able to teach the AI agent of combat behaviors of an expert player in a modified version the First Person Shooter game Soldier of Fortune 2.

In order to define the optimal behavior for the agent, the model extracts data by observing an expert player playing the game. A feature set of common actions is extracted from the data collected according to their importance. In this particular game, some actions would be move front/back, accelerate, jump, player's health status etc. Through this data extraction, the author is able to translate the data into feature vectors which will be used to train the Neural Network algorithm.

The major drawback of this method relies on the fact that the learning and behavior of the agent is directly based on the experience of the machine learner. An in inexperienced one will lead to an inexperienced AI agent. In addition, the model requires a huge amount of samples to map all possible behaviors and for the Neural Network to produce minimal errors.

A.2.2 Reinforcement Learning. Reinforcement Learning (RL)[2] is a type of machine learning, that allows machines and software agents to automatically determine the ideal behaviour within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behaviour, known as the reinforcement signal. In machine learning, the environment is typically formulated as a Markov decision process (MDP). The agent receives a reward, which depends on the action and the state. The goal is to find a function, called a policy, which specifies which action to take in each state, so as to maximize some function of the sequence of rewards. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

RL allows the machine or software agent to learn its behaviour based on feedback from the environment. This behaviour can be learned once and for all, or keep on adapting as time goes by. If the problem is modelled with care, some RL algorithms can converge to the global optimum, to module ideal behaviour that maximizes the reward.

There are some fundamental problems that RL must tackle. Firstly, it is often too memory expensive to store value of each state, since the problem can be pretty complex. Solving this involves looking into value approximation techniques, such as Decision Trees or Neural Networks. There are many consequence of introducing these imperfect value estimations, and research tries to minimize their impact on the quality of the solution. Secondly, problems are also generally very modular, as similar behaviours reappear often, and modularity can be introduced to avoid learning everything all over again. Another problem that usually occurs in large scale games, is the problem of delayed reward. For example an agent makes actions and receives rewards only at the end of the game. Then the problem that arises is that we don't know which of the actions was responsible for winning/losing the game. It is fundamentally impossible to learn the value of a state before a reward signal has been received. In large state spaces, random exploration might take a long time to reach a rewarding state. The only solution is to define higher-level actions, which can reach the goal more quickly. Finally, due to limited perception, it is often impossible to fully determine the current state.

An issue that arises as a consequence of learning is the problem of overfitting, which may occur if a game agent has learned to adapt its performance according to a very specific set of states from the game environment and remains unable to perform generalization, resulting in poor performance when new game states are encountered [11].

We will now review some noteworthy RL techniques used in recent videogames.

Deep Reinforcement Learning. Deep learning, is really just a term to describe certain types of neural networks and related algorithms that consume often very raw input data. They process this data through many layers of nonlinear transformations of the input data in order to calculate a target output.

Firoiu et al. [10] present a very similar approach to Mnih et al. [21] work but extend it to a more complex environment. More specifically they focus on Super Mario Bros Melee (SSBM), a more complex multi-player game. Its large and partially observable state, the transition dynamics and the delayed rewards pose a crucial challenge to their method. The multiplayer aspect of the game adds an entirely new dimension of complexity since success is no longer a single, absolute measure given by the environment, but instead must be defined relative to a variable, unpredictable adversary. In their method they use Q-networks similar to Minh's [21] work and exploit a set of policy gradient methods to map states to actions. The agent with the appropriate parameter tuning was able to defeat the opposing AI at its highest level and achieve similar rewards with those of human experts.

Motivated Reinforcement Learning. Another noteworthy approach is Merrick et al. [19] Motivated RL approach. In this approach the agent is able to explore the environment and learn new behavior in response to interesting experiences, allowing to display progressively evolving behavioural patterns. In more dynamic worlds the

agent is able to learn and adapt its behavior according to surrounding changes. Motivated reinforcement learning agents are meta-learners which use a motivation function to provide a standard reinforcement learning algorithm with an intrinsic reward signal that directs learning. Unlike existing non player character(NPC) technologies, the motivation function uses domain independent rules based on the concept of interest in order to calculate an intrinsic motivation signal. Skill development is dependent on the agent's environment and its experiences rather than on character or domain specific rules or state machines. This means that a single agent model applied to different NPCs will develop different skills depending on the NPC's environment. These skills are developed progressively over time and can adapt to changes in the agent's environment. The authors apply the MRL model to a number on NPC agents in a simple role playing game scenario in Second Life game. Given a set of rules and actions according to the role of the NPC (supporting or partner) they manage to teach the agent to adapt to existing behavioral patterns in dynamic worlds.

In a very similar approach Singh et al. [29] discuss about an evolutionary perspective of defining new optimal reward framework using Intrinsically Motivated RL. Their focus relies on generating the optimal fitness-based reward function for an agent to use. Both of those papers provide very interesting insights about the behavior of the agent in regards with the reward function used by the RL algorithm.

Inverse Reinforcement Learning. Inverse Reinforcement Learning (IRL) is the problem of recovering the underlying reward function from the behavior of an expert [8]. IRL problems work in the opposite way to reinforcement learning - when there is no explicitly given reward function, we can use an IRL algorithm to derive a reward function by observing an expert's behaviour throughout the MDP environment. Rewards are mapped to features within the states to reflect the importance of those features to the expert. This analysis of expert behaviour yields a policy that attempts to perform in a manner close to the expert [3].

Lee et al. [16] present a method that tries to imitate human like behavior from Super Mario, by observing the actions of a human expert player. During play sessions they calculate the player's behavior policies and reward functions by applying IRL to the player's actions in game. Through IRL they manage to define an optimal policy which results into performance similar to the human expert under specific conditions. This method is not considered in our work since we are not using human experts to teach our AI agent, but it could prove valuable work for future research.

A.2.3 Evolutionary Machine Learning. Xiao et al. [37], present an active learning framework for black box software testing that samples input and output pairs from a blackbox and learns a model of the system's behavior. The model is also used to select new inputs for sampling. Their approach is based on semi automated gameplay analysis by machine learning (SAGA-ML) applied to the popular Electronic Arts' FIFA 99 game. More specifically, the framework treats the game engine as a black box while SAGA-ML interacts with it through an abstraction layer that translates specific data and function calls to an abstract state format. The sampler uses the abstraction layer to to evaluate situations by running a sequence of actions and observing their results. The learner component utilizes the data from the sampler to construct a concrete model of the game's behavior. The sampler and earner components combined describe the learning part of the framework. In the end the learned model is given to the game-specific visualizer for the designer to evaluate. To evaluate the framework, the authors used rule based learning. They constructed various rules for the agents and build a game specific visualizer called SoccerViz to display the rules in an intuitive fashion. Furthermore, they developed a new active learning technique specifically for rule-based learning systems ,called decision boundary refinement sampling (DBRS) which aims to visualize the rules in forms of region and rectangles to evaluate whether they overlap.

Chan [7] proposes an evolutionary learning approach of agent behavior to improve testing in commercial games, based on evolving user action sequences producing unwanted or to be tested behavior. The approach allows for measuring how near a sequence of game states comes to unwanted behavior and use the measures within fitness function in genetic algorithm. The author test the proposed method in the famous commercial game FIFA 99.

To begin with, the method takes the inputs of the player and translates them as action commands given to the game out of a set of possible actions, called action sequence. Then an action sequence is executed n times to result into k state sequences. From the set of sequences produced the method is able to evaluate the behavior of the agent. The genetic algorithm is then used to create action sequences that produce an unwanted behavior specified by the fitness function in more than a certain percentage of runs of the game using the action sequence as user interaction input. The GA algorithm works on the set F of sequences of indexes for actions. The evaluation of the fitness of an individual is based on k runs of this individual as input to the game from a given start game state s_0. Associating a low fitness-value with a good individual leads to unwanted game behavior. The above method is tested in FIFA 99, where the author is trying to determine the behavior of the agent in near goal situations. Action sequences/rules are being integrated into the state sequence like the distance of the agent position from the goal, whether the agent has the ball and so on. The sequence run 100 times and 80 of them it resulted to goal.

A.3 Path Planning

Path planning is a crucial ingredient in videogames since entities like players and NPC's must find their routes to several locations [25]. Path planning is a challenging problem to solve for a number of reasons. Paths must avoid obstacles and be relatively short, leading to a difficult combinatorial problem in an environment that consists of a huge number of obstacles. Especially in recent vast open-world video games when entities have multiple degrees of freedom, the dimension of the space of motion becomes larger, adding to the difficulty.

In this section we will discuss one of the most popular path planning approaches in video games. In addition we will review a novel approach of path planning with the use of reinforcement learning methods discussed above.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the

goal node[1]. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n) \tag{2}$$

where n is the last node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristicthat estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissable, meaning that it never over-estimates the actual cost to get to the nearest goal node. A* algorithm is the most commonly used path planning algorithm in games due to its simplicity and effectiveness. In recent years several other techniques have emerged like navigation meshes, probabilistic roadmaps and real time path planning. However we will not dig more into path planning literature as it is not the main subject of our research. We will only discuss how path planning can be intergated into a game with help from AI learning algorithms.

Mirowski et al. [20] discuss about agent navigation in complex environments with the use of reinforcement learning techniques. They show that data efficiency and task performance can be dramatically improved by relying on additional auxiliary tasks leveraging multimodal sensory inputs. They implement a combination of learning the goal-driven RL problem with auxiliary depth prediction and loop closure classification tasks. This approach can learn to navigate from raw sensory input in complicated 3D mazes, approaching human-level performance even under conditions where the goal location changes frequently. Of course learning to navigate through RL methods in partially observable environments is not a trivial task. Rewards have to be sparsely distributed in the environment where there might be only one target location. In addition, in cases of dynamic elements in the environment the agent should use memory to redefine the goal location, visual observation etc. The authors use depth prediction to extract data of the visual observation of the agent and the 3D geometry of the environment. Then they directly invoke a loop closure technique that trains the agent to predict if the current location has been previously visited within a local trajectory. Finally their results and analysis highlight the utility of self-supervised auxiliary objectives, namely depth prediction and loop closure, in providing richer training signals that bootstrap learning and enhance data efficiency. This is a very novel approach that might be proven very useful in planning the trajectory of the agent.

A.4 Additional Information

In this section we provide more information about the technical parts of the implementation of our algorithm. An overview of the algorithm that we used is shown in Figure 7.

In every game tick we get an instance of the world state and check whether we got a new observation since last check. Whenever we have new observations, we store the necessary information into a json file and assign it to an object. In this case this is done by the command in line 296. Now we can extract the data of the observations and feed the agent with useful information about the world. For example we can extract the "yawDelta" which represents the angle of which the agent should turn in order to look directly at a specific position. We then use this information to send a command through the framework to the agent and dictate where and how it should turn in order to steer himself towards a specific

```
291     # main loop:
292     while reward < reward_goal and world_state.is_mission_running and flag == 0:
293         world_state = agent_host.getWorldState()
294         if world_state.number_of_observations_since_last_state > 0:
295             msg = world_state.observations[-1].text
296             ob = json.loads(msg)
297             yaw = ob.get(u'Yaw', 0)
298             if u'yawDelta' in ob:
299                 current_yaw_delta = ob.get(u'yawDelta', 0)
300                 agent_host.sendCommand( "turn " + str(current_yaw_delta) )
301                 agent_host.sendCommand( "move " + str(1.0 - abs(current_yaw_delta)) )
302             if world_state.number_of_rewards_since_last_state > 0:
303                 delta = world_state.rewards[-1].getValue()
304                 print "Reward: " + str(delta)
305                 reward += delta
306             if reward >= reward_goal:
307                     SetVelocity(0)
308                     SetTurn(0)
309                     flag = 1
310                     agent_host.sendCommand("hotbar.1 1")
311                     agent_host.sendCommand("hotbar.1 0")
312                     time.sleep(2)
313                     key = ob['InventorySlot_'+str(0)+'_item']
314                     while ob['InventorySlot_'+str(0)+'_size'] > 0:
315                         print "discarded: " + ob['InventorySlot_'+str(0)+'_item']+ "  size:" + str(ob['InventorySlot_'+str(0)+'_size'])
316                         agent_host.sendCommand("discardCurrentItem")
317                         ob['InventorySlot_'+str(0)+'_size'] -= 1
318                     time.sleep(1)
319                     world_state1 = agent_host.getWorldState()
320                     msg1 = world_state1.observations[-1].text
321                     ob1 = json.loads(msg1)
322                     print "Crafting pumkin pie....."
323                     agent_host.sendCommand("craft pumpkin_pie")
324                     time.sleep(2)
325                     world_state3 = agent_host.getWorldState()
326                     if world_state3.number_of_rewards_since_last_state > 0:
327                         reward1 = world_state3.rewards[-1].getValue()
328                         #print "Reward1: " + str(reward1)
329                     if reward1 == 1000:
330                         UpdateRewardMatrix(key,0)
331                         print "Item discarded not needed"
332                         print "Crafting pumkin pie succeeded....."
333                         del recipe[key]
334                         success = 1
335                     if reward1 == 0:
336                         UpdateRewardMatrix(key,1)
337                         print "Item discarded needed"
338                         print "Crafting pumkin pie failed....."
339
340         time.sleep(0.1)
```

Figure 7: Example code of the main loop.

```
1   20180318T152307.782348 0:0.9940052
2   20180318T152309.905470 0:1.0
3   20180318T152310.075479 0:1.0
4   20180318T152311.148541 0:2.0
5   20180318T152312.004590 0:1.0
6   20180318T152313.376668 0:0.9940052
7   20180318T152315.082766 0:0.9940052
8   20180318T152315.851810 0:0.9940052
9   20180318T152318.126940 0:2.0
10  20180318T152318.650970 0:1.0
11  20180318T152319.569022 0:1.0
12  20180318T152320.261062 0:1.0
13  20180318T152321.483132 0:1.9940052
14  20180318T152323.673257 0:1.0
15  20180318T152325.305350 0:1.0
16  20180318T152326.607425 0:1.0
17  20180318T152326.853439 0:1.9940052
18
```

Figure 8: Rewards output file.

coordinate. This is shown in lines 298-301. In line 306 we check if we reached the reward goal that we have set from before, so this serves as a condition to start checking the functionality of the crafting command. We command the agent to stop moving and receiving new observations and we perform the elimination process. We toss the first item on the inventory hotbar and we check whether we can still craft the desired item. This process is shown in lines 313-323. WE now ask for a new world state and compare it with the previous one. Whenever the crafting command succeeds and we get the desired item in our inventory, we have get a reward for receiving this new item. If in our new state we get this reward, we understand that the functionality of the command works and we update the reward matrix accordingly (lines 326-338).

The framework provides us with a functionality to store useful information of the mission is separate files. Specifically, it records the game and outputs a video of the mission. Additionally, in stores all the rewards that from the observations and the timestep in a

```xml
<ServerSection>
  <ServerHandlers>
    <FlatWorldGenerator destroyAfterUse="true" forceReset="true" generatorString="3;7,220*1,5*3,2;3;,biome_1" seed=""/>
    <DrawingDecorator>
      <DrawCuboid colour="RED" face="UP" type="carpet" x1="-50" x2="50" y1="226" y2="226" z1="-50" z2="50"/>
      <DrawItem type="egg" x="-18" y="250" z="30"/>
      <DrawItem type="egg" x="-26" y="250" z="10"/>
      <DrawItem type="pumpkin" x="-18" y="250" z="23"/>
      <DrawItem type="egg" x="11" y="250" z="19"/>
      <DrawItem type="fish" x="-16" y="250" z="15"/>
      <DrawItem type="sugar" x="3" y="250" z="-3"/>
      <DrawItem type="fish" x="0" y="250" z="-10"/>
      <DrawItem type="sugar" x="21" y="250" z="3"/>
      <DrawItem type="egg" x="12" y="250" z="-7"/>
      <DrawItem type="egg" x="-14" y="250" z="-24"/>
      <DrawItem type="pumpkin" x="-24" y="250" z="18"/>
      <DrawItem type="egg" x="-5" y="250" z="26"/>
      <DrawItem type="egg" x="-12" y="250" z="27"/>
      <DrawItem type="fish" x="-1" y="250" z="-25"/>
 <AgentHandlers>
   <ObservationFromFullInventory/>
   <ObservationFromSubgoalPositionList>
     <Point description="ingredient" tolerance="1" x="-18" y="227" z="30"/>
     <Point description="ingredient" tolerance="1" x="-26" y="227" z="10"/>
     <Point description="ingredient" tolerance="1" x="-18" y="227" z="23"/>
     <Point description="ingredient" tolerance="1" x="11" y="227" z="19"/>
     <Point description="ingredient" tolerance="1" x="-16" y="227" z="15"/>
     <Point description="ingredient" tolerance="1" x="3" y="227" z="-3"/>
     <Point description="ingredient" tolerance="1" x="0" y="227" z="-10"/>
     <Point description="ingredient" tolerance="1" x="21" y="227" z="3"/>
     <Point description="ingredient" tolerance="1" x="12" y="227" z="-7"/>
     <Point description="ingredient" tolerance="1" x="-14" y="227" z="-24"/>
     <Point description="ingredient" tolerance="1" x="-24" y="227" z="18"/>
     <Point description="ingredient" tolerance="1" x="-5" y="227" z="26"/>
     <Point description="ingredient" tolerance="1" x="-12" y="227" z="27"/>
     <Point description="ingredient" tolerance="1" x="-1" y="227" z="-25"/>
   <RewardForCollectingItem dimension="0">
     <Item distribution="" reward="1000" type="pumpkin_pie"/>
     <Item distribution="" reward="1" type="sugar"/>
     <Item distribution="" reward="0.994005211847" type="egg"/>
     <Item distribution="" reward="1" type="pumpkin"/>
     <Item distribution="" reward="1" type="fish"/>
     <Item distribution="" reward="0" type="beef"/>
     <Item distribution="" reward="0" type="chicken"/>
     <Item distribution="" reward="0" type="potato"/>
   </RewardForCollectingItem>
   <ContinuousMovementCommands turnSpeedDegs="240"/>
   <InventoryCommands/>
   <ChatCommands/>
   <SimpleCraftCommands/>
   <MissionQuitCommands quitDescription=""/>
```


Figure 9: Example of MissionInit output.

text file called "rewards.txt" as seen in Figure 8. Finally,it provides an output of all the necessary information about the mission in an XML format, such as the position of every item in the world, the observations and the rewards. An example is shown in Figure 9.

REFERENCES

[1] [n. d.]. A* search algorithm. ([n. d.]). https://en.wikipedia.org/wiki/A*_search_algorithm.

[2] [n. d.]. Reinforcement Learning. ([n. d.]). http://reinforcementlearning.ai-depot.com/.

[3] Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*. ACM, 1.

[4] Robert V Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[5] Christian Buhl and Fazeel Gareeboo. [n. d.]. Automated testing: a key factor for success in video game development. Case study and lessons learned.

[6] Alex Castrounis. 2017. Artificial intelligence, deep learning and neural networks. (2017). https://estidevelopers.com/2017/07/08/neuralnetworks-deeplearning/.

[7] Ben Chan, Jörg Denzinger, Darryl Gates, Kevin Loose, and John Buchanan. 2004. Evolutionary behavior testing of commercial computer games. In *Evolutionary Computation, 2004. CEC2004. Congress on*, Vol. 1. IEEE, 125–132.

[8] Jaedeug Choi and Kee-Eung Kim. 2011. Inverse reinforcement learning in partially observable environments. *Journal of Machine Learning Research* 12, Mar (2011), 691–730.

[9] Alexander Elyasov, I.S.W.B. Prasetya, Jurriaan Hage, and Andreas Nikas. 2014. Reduce first, debug later. In *Proceedings of the 9th International Workshop on Automation of Software Test*. ACM, 57–63.

[10] Vlad Firoiu, William F Whitney, and Joshua B Tenenbaum. 2017. Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning. *arXiv preprint arXiv:1702.06230* (2017).

[11] Leo Galway, Darryl Charles, and Michaela Black. 2008. Machine learning in digital games: a survey. *Artificial Intelligence Review* 29, 2 (2008), 123–161.

[12] Ben Geisler. [n. d.]. Integrated machine learning for behavior modeling in video games.

[13] HU Huixian and LU Lu. 2016. Automatic Functional Testing of Unity 3D Game on Android Platform. (2016).

[14] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. 2015. An automated model based testing approach for platform games. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 426–435.

[15] Mikko Lahti. 2015-02-09. *Game Testing in Finnish Game Companies; Pelitestaus suomalaisissa peliyrityksissÃď*. G2 Pro gradu, diplomityÃű. http://urn.fi/URN:NBN:fi:aalto-201502191908

[16] Geoffrey Lee, Min Luo, Fabio Zambetta, and Xiaodong Li. 2014. Learning a super mario controller from examples of human play. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*. IEEE, 1–8.

[17] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. 2010. What Went Wrong: A Taxonomy of Video Game Bugs. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games (FDG '10)*. ACM, New York, NY, USA, 108–115. https://doi.org/10.1145/1822348.1822363

[18] Jialin Liu, Julian Togelius, Diego Pérez-Liébana, and Simon M Lucas. 2017. Evolving Game Skill-Depth using General Video Game AI Agents. *arXiv preprint arXiv:1703.06275* (2017).

[19] Kathryn Merrick and Mary Lou Maher. 2006. Motivated reinforcement learning for non-player characters in persistent computer game worlds. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*. ACM, 3.

[20] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. 2016. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673* (2016).

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[22] Kshirasagar Naik and Priyadarshi Tripathy. 2011. *Software testing and quality assurance: theory and practice*. John Wiley & Sons. 16–18 pages.

[23] Alfredo Nantes, Ross Brown, and Frederic Maire. 2008. A Framework for the Semi-Automatic Testing of Video Games.

[24] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11.

[25] Mark H Overmars. [n. d.]. Path planning for games.

[26] I.S.W.B. Prasetya. 2015. T3i: a tool for generating and querying test suites for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 950–953.

[27] I.S.W.B. Prasetya. 2018. Temporal algebraic query of test sequences. *Journal of Systems and Software* 136 (2018), 223–236.

[28] C.P. Schultz and R.D. Bryant. 2016. *Game Testing: All in One*. Mercury Learning & Information. https://books.google.nl/books?id=jwsyjgEACAAJ

[29] Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. 2010. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development* 2, 2 (2010), 70–82.

[30] Mike Treanor, Alexander Zook, Mirjam P Eladhari, Julian Togelius, Gillian Smith, Michael Cook, Tommy Thompson, Brian Magerko, John Levine, and Adam Smith. 2015. AI-based game design patterns. (2015).

[31] Hiroto Udagawa, Tarun Narasimhan, and Shim-Young Lee. 2016. *Fighting Zombies in Minecraft With Deep Reinforcement Learning*. Technical Report. Technical report, Stanford University.

[32] Tanja EJ Vos, Paolo Tonella, I.S.W.B. Prasetya, Peter M Kruse, Onn Shehory, Alessandra Bagnato, and Mark Harman. 2013. The FITTEST tool suite for testing future internet applications. In *International Workshop on Future Internet Testing*. Springer, 1–31.

[33] Tanja EJ Vos, Paolo Tonella, Joachim Wegener, Mark Harman, I.S.W.B. Prasetya, Elisa Puoskari, and Yarden Nir-Buchbinder. 2011. Future internet testing with fittest. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 355–358.

[34] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[35] Laurie Williams. 2006. Testing Overview and Black-Box Testing Techniques. (2006).

[36] Ping-Hung Chen Chia-Sheng Hsu Woei-Kae Chen, Chien-Hung Liu. 2016. A Game Framework SupportingAutomatic Functional Testing for Games. Springer.

[37] Gang Xiao, Finnegan Southey, Robert C Holte, and Dana Wilkinson. 2005. Software testing by active learning for commercial games.