AUTOMATED PERFORMANCE TESTING FOR VIRTUAL REALITY

Georgios Kokkinos 5793459

A thesis presented for the degree of
Game and Media Technology

Game and Media Technologies
Utrecht University
Netherlands
January 2018

Preface

First and foremost, this thesis is dedicated to my family for supporting me. Without them I wouldn't have the opportunity to join this university!

Huge thanks to my supervisor, Whisnu Prasetya for his guidance through the journey of completing this research. My thanks also go to my second supervisor Frank Dignum for the feedback and evaluation of my thesis.

Thanks to Patryk Wroblewski and ForceFieldVR for accepting me as a QA tester for a small period of time as an intern, where I learned the in and outs of manual testing for virtual reality worlds. Without this experience the research would be lacking.

Automated Performance Testing for Virtual Reality

Master Thesis

Georgios Kokkinos*
Utrecht University
g.kokkinos@students.uu.nl

S.W.B. Prasetya†
Utrecht University
Utrecht, Netherlands
S.W.B.Prasetya@uu.nl

F.P.M. (Frank) Dignum‡
Utrecht University
Utrecht, Netherlands
F.P.M.Dignum@uu.nl

ABSTRACT

With the coming of Virtual Reality (VR) and its increased computational requirements, performance testing has become extremely demanding. In this study automation of such testing is investigated by comparing results from manual testers versus adaptive agents that play through a First Person Shooter (FPS) game. Results indicate that automation of performance testing can provide good results and is on par with manual testers in several aspects. Unfortunately we can't conclude that such an automation as described here can replace the human tester but it can certainly help ease the workload.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Interactive games*; **Software performance**; • **Computing methodologies** → *Artificial intelligence*; *Machine learning*;

KEYWORDS

Behavior trees, Finite state machine, Reinforcement Learning, Unreal Engine, Virtual Reality, Head Mounted Display, First Person Shooter, Frame per second, Domain specific language.

ACM Reference Format:
Georgios Kokkinos, S.W.B. Prasetya, and F.P.M. (Frank) Dignum. 1997. Automated Performance Testing for Virtual Reality: Master Thesis. In *Proceedings of Test conference (TEST '18).* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Virtual reality (VR) industry noticed a huge surge in popularity from 2015 and onwards. It has a big range of uses from entertainment and training to innovative forms of marketing. Virtual reality headsets have been released from major companies like Oculus from Facebook and Vive from HTC and Steam. Samsung released the Gear VR to make VR possible on their smartphones, but with Google Cardboard developers were also able to create VR experiences for other types of smartphones [39]. The game industry has taken advantage of this fairly new technology and a lot of companies, indie and major ones started developing experimental games ,tech demos and even AAA (triple -A) titles. VR gaming is on the rise with 63 % of most frequent gamers being familiar with it and one out of three stating that they will buy VR products in the next year [16]. Resources spent for the creation of such software are immense and failure is not an option.

Extensive testing has to be done in order to assure that the quality of the project is high enough to be competitive in the market. The game development process is mostly based on black box testing. It happens though, that game testing can become more complex as a result of the diverse development for different components of the game and automation is mandatory to achieve the required results. Testing plays an important role in every development steps, and the developing team only moves to the next level after approval from the testing team [6].

With the coming of VR the range of issues applicable for testing expands, with the most notable ones being bigger frame rate dependencies ,motion sickness and tracking [45]. When the frame rate drops below 60 or else the rendering and game logic requirements exceed 20ms of processing time, jittering can be noticed making the experience "laggy" causing discomfort to the user [47] [45]. As a result being able to achieve this frame cap consistently and give the user a seamless experience, extensive testing has to be done with full playthroughs in all parts of the game. If you combine it with diverse hardware (desktop or mobile) that can have different impact in the game, the resources required for testing became immense. Our core idea is to develop an AI-agent that will provide essential data in performance runs so we can show that automation is possible and successful.

AI has been used before to play through a game with different goals in mind. There are various AI competitions about different games where competitors create bots that play the game in order to beat it or compete between them . One such competition is the Infinite Mario Competition with the intention to win as many levels as possible. The clear victor in the competition was the A* algorithm [40], which due to Mario's deterministic nature, allowed for perfect prediction and timing. One interesting point was that the algorithm, played in a manner that performs impossible feats, appearing totally unnatural to the human eye. Competitions with different goals in mind exist like the Turing test track of Mario AI Championship with intentions of playing the game in a way similar to a human. The most common methods used in this competition

*Master student
†First supervisor
‡Second Supervisor

TEST '18, January 2018, Utrecht, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

were hand coded rules, supervised learning , indirect representation, neuroevolution via Artificial Neural Networks (ANN), directed scripting and behavior trees [20].

As mentioned already , time and money requirements for game development are huge. All of the modern game companies use Game Engines , which are state-of-the-art resource for game development. They support common foundations such as texture rendering, world management, event handling and AI. Even though game engines provide all the core functionality needed , each game is its own entity with different logic than others and has to be treated according to its needs. Same goes for its AI. Different game logic has to be implemented according to in game situations and circumstances although some parts may be common across different games (e.g path planning). It works in a similar way if you want to create a bot that plays through the game.

In this study we develop a VR game and implement an agent that can play through it. We propose a basic agent implementation that can potentially work in multiple environments without major changes in the code. We also provide a special implementation of the agent that can provide good results for re-testing purposes. We perform a study where we evaluate manual testers versus our implemented agent in terms of performance testing findings. Results indicate that automation of performance testing can provide good results and is on par with manual testers in several aspects. The paper proceeds as follows: In section 2 we provide and analyze related work. In section 3 we show our research goals. Next in section 4 we take a look at our research approach. Experiments and discussion are provided in section 5. Last in section 6 we conclude our paper.

2 RELATED WORK

Work has been done before to automate testing in the video games industry. One approach that is popular in software testing in general is the record and playback testing. Such a method is presented on "A Game Framework Supporting Automatic Functional Testing for Games" [3] where they create an HTML5 testing framework for games. They basically enhanced that framework with a testing layer that can execute (playback) test scripts that perform user events and assert the correctness of the game. A tester can create a test script either by writing the script directly or by playing the game and capturing the game-play actions. The test script consists of various features such as controller events and requests to execute any methods / functions of the game. (e.g. load a certain level). One major disadvantage of record and playback methods is that manual work is required to either write the script that will run the game or record a certain number of actions that will create the script automatically. Furthermore another difficult situation to handle, is the timing of events. Due to the dynamic nature of games (e.g enemies, randomization of events e.t.c) when timing is not exactly right as in the recorded test script it may not always reproduce the same results, which is unacceptable. As a result adaptation in real time is required.

There has also been substantial work in the domain of creating Domain specific Languages (DSL) for games that can be used to express a game as an application domain in a higher level of abstraction. An important mention in this domain is the Game Description Language (GDL) which was first introduced by the Stanford Logic Group of Stanford University [32]. The idea behind it is to create general game playing which refers to the design of Artificial Intelligence programs that are able to play more than one game successfully. For many games like chess, computers are programmed to play these games using a specially designed algorithm, which cannot be transferred to another context. For example, a chess playing computer program cannot play checkers. A General Game Playing system, if well designed, would even be able to help in other areas, such as in providing intelligence for search and rescue missions.

In the same spirit, we created our Agent and tried to generalize functionalities that possibly remain consistent in the majority of VR games. We didn't create a DSL but if we hypothetically connect it to one, it will provide flexibility in the manner of our Agent's suitability to play on other games. The genre of the game also plays an important part when you try to create a higher level abstract description. We performed a quantity measurement of how many VR games are first person games and by first person we refer to the graphical perspective is rendered from the viewpoint of the player's character. It turned out that 75% are first person games so we decided to develop such a game and perform our tests in that regard.

For the agent part we used a mix of behavior trees, dynamic programming and reinforcement learning for retesting. A brief explanation of the technologies is described here while more information about them and extra related work about the most common machine learning methods for video games [26] including Neural Networks [19], Reinforcement Learning [37], Apprenticeship Learning[36] and Evolutionary methods [21] [4] can be found in the appendix.

A behavior tree (BT) is a mathematical model of plan execution used in computer science, robotics, control systems and video games. They describe switchings between a finite set of tasks in a modular fashion. They have seen great success in the creation of AI for video games. Considering as the evolution of finite state machine (FSM) with the advantage over them to be more easily manageable as the AI grows in complexity and also being reusable. Each tree is goal oriented, meaning that they are associated with a high level goal they want to complete. The leaves are the actual commands that control the AI entity, and forming the branches are various types of utility nodes that control the AI's walk down the trees to reach the sequences of commands best suited to the situation. The trees can be extremely deep, with nodes calling sub-trees which perform particular functions, allowing for the developer to create libraries of behaviors that can be chained together to provide very convincing AI behavior. Development is highly iterable, where you can start by forming a basic behavior, then create new branches to deal with alternate methods of achieving goals, with branches ordered by their desirability, allowing for the AI to have fall back tactics should a particular behavior fail. [5] [48]

In our research we tried to make the agent as adaptive as possible to the environment. One major influence from the literature is the work Agent Architecture Considerations for Real-Time Planning in Games. This paper describes lessons learned while implementing real-time planning for NPCs for F.E.A.R., a AAA first person shooter shipped for PC in 2005 [34].

They implement their agents using event driven methods and define what the agent perceives as stimulus of the environment. They also create an adaptive system based on a data structure that is stored dynamically. They call it WorkingMemoryFact which is a record containing a set of associated attributes. Different subsets of attributes are assigned depending on the type of knowledge the fact represents. Ten possible types of knowledge are available, including Character, Object, Disturbance, Task, PathInfo, and Desire Facts.

A pseudo-code representation of a WorkingMemoryFact follows found in the above mentioned paper:

```
WorkingMemoryFact
{
        Attribute <Vector3D> Position
        Attribute <Vector3D> Direction
        Attribute <StimulusType>Stimulus
        Attribute <Handle> Object
        Attribute <float> Desire
        ...
        float fUpdateTime
}

Attribute <Type>
{
        Type Value
        Float fConfidence
}
```

Next they introduce the confidence value. Confidence may represent an NPC's current degree of stimulation, proximity to some object, or degree of desire. When applied to the Stimulus attribute, confidence represents how confident the NPC is that he is sensing some stimulus. For example, the confidence of a Character Fact's Stimulus attribute indicates the current level of visual stimulation that the NPC is aware of for this character. The intensity of an NPC's Desire attribute is characterized by his confidence that he is feeling this desire. The confidence value of a Desire Fact's Desire attribute indicates the NPC's current urge to satisfy some desire. This could be for example a value of how much he wants to attack an enemy and is changed dynamically based on stimulus.

With performance testing comes a lot of testing that needs repeating for different devices and most important a load of regression. When we are doing retesting we need to also prioritize our test cases to get feedback faster. Our core idea for the agent is to reach the places that performed worse in previous runs faster and as a result identify possible performance bugs as soon as possible. For the purpose we used reinforcement learning to memorize the regions with low performance drops and prioritize them using path nodes that lead to them faster.

Our major influence for that was the work Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration [42]. Their method uses reinforcement learning (RL) to select and prioritize test cases according to their duration, previous last execution and failure history. In RL, an agent interacts with its environment by perceiving its state and selecting an appropriate action, either from a learned policy or by random exploration of possible actions. As a result, the agent receives feedback in terms of rewards, which rate the performance of its previous action. [7] A state represents a single test case metadata, consisting of the test cases approximated duration, the time it was last executed and previous test execution results. As an action the test case priority is returned. After all test cases in a test suite are prioritized, the prioritized test suite is scheduled, including a selection of the most important test cases, and submitted for execution. With the test execution results, i.e., the test verdicts, a reward is calculated and fed back to the agent. From this reward, their agent adapts its experience and policy for future actions. We adapted this method in a VR world by mapping certain regions as our test case targets.

Another important step in our literature is the work reinforcement learning for solving shortest - path and dynamic scheduling problems. They show that an RL-based implementation of the internet protocol with adaptive and cooperative agents can transport data following the shortest route in time from a source computer to a destination, even if the transfer capacities of connections change in time. In contrast to the famously used static routing algorithms, the proposed method is capable to cope with dynamic conditions as well. More on this on the section 6, where we discuss the advantages of such a method for dynamic environments.

To finalize our related work section we take a look at two works about automation in more traditional software testing. First , in the work "Unit Testing Tool Competition - Round Three" [38], the authors evaluate a number of recent automated unit testing tools, using state of the art techniques. More specifically they evaluate seven tools that generate JUnit test cases for Java classes. The results are similar to what we stated in the abstract, namely that so far automated tools cannot completely replace humans. There are errors that were found by human and tools; so for these at least theoretically they can replace human. Nonetheless there are errors that humans found and the tools did not, exactly like our approach. Our implementation features a knowledge base that gives information to our agent about the virtual world. It can be used to give greater importance to certain objects or decide what actions are to be performed with them. It can possibly be considered as a way to create an abstract test case and if it's used for this purpose, creating tools to automate test case generation like in the mentioned methods of this paper, would provide interesting future work.

Next we discuss the work "Future Internet Testing with FITTEST" [44]. The FITTEST project, was set to explore new testing techniques that will improve the capacity to deal with the challenges of testing Future Internet applications. Future Internet is a general term for research activities on new architectures for the Internet and feature different challenges in testing. For example about large scale and evolution of components which means that components are continuously added to the system and evolve in a rapid way, they propose prioritization of tests according to available time and budget. In the same way we propose a method to overcome a different aspect of performance testing. In our paper instead of regular playthroughs for finding performance errors, we also proposed a method that provides better performance on retesting with prioritizing of certain regions. Another proposal of theirs is about large Feature-configuration space where highly customizable environments offer too many variables that can be tweaked and need testing. This is highly related to video games as they are considered environments with possible infinite actions/variables and having a

test case for each different component separately can be extremely time consuming. In our work the abstract test cases of our knowledge base can lead to add more methods into our implementation for example about combinatorial testing (CT) which the authors of FITTEST also propose, as a method of software testing that for each pair of input parameters of a system, tests all possible discrete combinations of those parameters[2]. More about CT can be found in the related section of the appendix. Creating a paper such as this could be possible if our research improves and expands into covering different aspects of testing.

3 RESEARCH GOALS

Due to the different nature of the VR games that require a headset instead of a monitor and looking around with head movement instead of a mouse or a controller extra challenges appeared when developing agents for playthroughs. Also to the best of our knowledge automation in performance testing of VR games was not covered before in the academic literature so our research question is formulated as follows:

- **RQ1**: Can agent-based AI be useful for performance testing in VR worlds?

Agent-based AI is a broad term and an important part of the above question would be:

- **RQA:** What is the intelligence level and behaviours needed to playthrough the game with performance testing as a goal?

This basically breaks down to if the agent should behave as a human or not. Each approach has its advantages and disadvantages as we will see later in the discussion section where we evaluate manual testers versus our agent that behaves more in a way that resembles what a tester would do.

Furthermore we will evaluate if prioritization of test cases can be learned and how a reinforced path planning routing will compare versus a manual tester in terms of time required to find performance issues. By that we will answer the our second research question:

- **RQ2**: Can we retest performance issues using reinforcement learning in VR?

The paper proceeds as follows: first in section four we provide insight into our implementation of the adaptive agents. Next in section five we present the experiment setup and the results. Last in section six we conclude and offer some discussion points.

4 RESEARCH APPROACH

For this study a VR First person shooter (FPS) was designed in a way that it resembles modern video games. It features different regions with enemies, obstacles and items that can be interacted with.

Given a VR under test, our approach works by deploying an autonomous test agent to the VR in the role of a player. Such an agent is configured to have a certain testing goal, e.g. to find performance issues in the VR. Once deployed, it will autonomously explore and interact with the VR towards fulfilling its goal. In theory multiple test agents can be deployed, but in the current work we limit ourselves to single agent setups. Since in this context there is only one agent, we will refer to it by "the agent".

The agent is based on a sensing system of "seeing" and "hearing" and can adapt in a dynamic environment while also different variations of the game without altering the code behind it. For the path planning we use navigation meshes[43].

For the cause Unreal Engine 4.16.3 (UE4) was used. It's a powerful game engine that's been used in a variety of successful AAA games. With its code written in C++, UE4 features a high degree of portability. It has won several awards, including the Guinness World Records award for "most successful video game engine". Furthermore it features some basic AI architectures like Behavior trees and Navigation meshes. For the implementation of the agent logic a mix of Blueprints, C++ and Python was used. The Blueprints Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. Next we will present more in depth details about the agent implementation.

4.1 Basic Agent Information

For the agent to work with a VR headset a certain implementation step is crucial. We need to lock the VR headset orientation settings and not allow strafing and turning of the camera with movement from the headset. This will allow the camera to follow the agent without outside hindrances. The orientation of the camera is purely controlled by where the agent focuses inside the game.

Next about the sensing system, the agent uses an overlapping cone originating from it's head and moving in the forward direction up to 1200 Unreal Metric Unit with a radius of 95 degrees. This serves for the vision function , meaning anything that overlaps with the cone will be perceived from the agent. For the hearing function an overlapping sphere is used that perceives any item that produces a sound in a certain radius. This is used for "hearing" the testing milestones that are used for retesting. (These "sensing" options are provided by the engine and can be seen in figure 1)

Our agent also features a global knowledge system that gives additional information about the objects that we want to interact with. They are used as a data structure in the following format:

```
{
        "Name": "0",
        "ActorName": "BP-Door_C",
        "ActionMapped": "Interact",
        "Ignore": false,
        "Desire": 0.6
    },
    {
        "Name": "1",
        "ActorName": "BP-Statue_C",
        "ActionMapped": "Interact",
        "Ignore": false,
        "Desire": 0.3
    },
}
```

It features a struct attribute "ActionMapped" that defines what should be done with an object when seen. Available options are:

(1) **Interact**, which basically performs the action that is specified in the game for the specific object (e.g "Open" for the door, "Collect for the statue" e.t.c.).
(2) **Wait**, which moves near the item and observes it.
(3) **Shoot**, which moves in a specific range and shoots at the object.

Fig 1. Example of sensing tools for the agent

Next it features a bool attribute "Ignore" that informs the agent if an item should be ignored or not . Last it features a float attribute "Desire" that gives a metric of how much the agent wants to interact with an item. This is used for prioritization reasons in case of multiple items being perceived simultaneously.

4.2 In depth look

More in depth about the implementation, when the agent sees or hears an object it updates a data structure called "ActorScores" that contains information about the identity of the object and its associated score value. The score value is a combination of the desire value from the global knowledge base with the distance between the agent and the object. The further away the object is, the lower the score is. As a result our agent moves to the object with the highest score first.

For the gameplay behaviours and most of basic Agent behaviour the Blueprint based language of Unreal Engine is used. A small example of the update of "ActorScores" can be seen on figure 2.

The node-function "GetScoreForActor" returns the score value for the object and the boolean value "Ignore" that states if it should be ignored or not. The "Branch" node is basically an IF statement that checks the previously mentioned boolean and if false, the object is added to the data table "Actor scores" together with its associated score using the "Add" node. Some extra details about the function "GetScoreForActor"; we basically read the "Desire" value from the knowledge base and get the difference between this value and the distance of the agent of the object as mentioned before. Because the desire values has a range between 0 and 1 we also normalize the distance value. As follows:

$$S = D - ||\vec{P} - \vec{O}||$$

With 'S' showing score ,'D' desire , 'P' vector of player location and 'O' vector of object location.

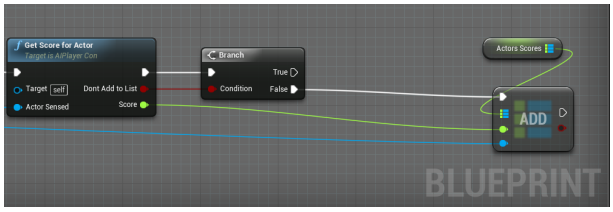

Fig 2. Blueprint Example

When no items are seen by the agent a grid of random size (between a certain radius) is created around the agent and he chooses a point for his new location goal that satisfies the criteria:

(1) It is far away from other points that the agent previously visited (Notated as DPV).
(2) It is far away from other objects that the agent previously visited (DOV).
(3) Its is far away from it's current location (DS).

A score function is formulated from the above criteria with specific multipliers that represent their importance:

$$GoalScore = 10 * DPV + 3 * DOV + 2 * DS \qquad (1)$$

We refer to this as equation (1). If any items are seen during the path , the action /movement is interrupted and the agent moves towards the newly discovered item.

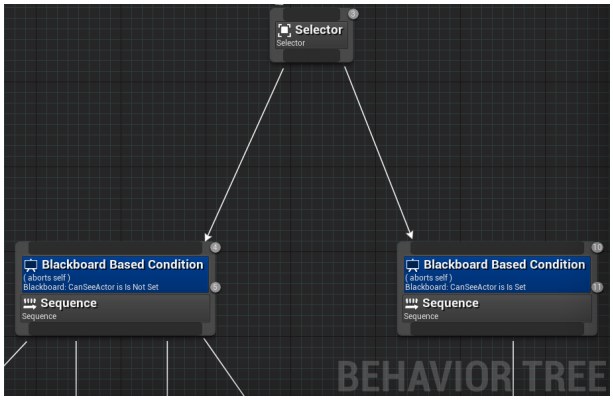

Fig 3. Behavior tree Example

On the behaviour tree this is represented with a selector node and a boolean check. As we talked about the BT's on the related work, selector is basically an if statement that checks a condition. The implementation of BT's in UE4 follows an event driven method. It is based on event-driven programming , a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads [8]. The behavior trees just passively listen for events which can trigger changes in the tree. When this happens a certain part of the tree can be executed (More on this on the related section of the Appendix). Basically when an item is perceived we set a bool value name "CanSeeActor" to true that will change the sequence of actions for the agent in the next tree traversal.

C++ was used for more advanced requirements of the project when the blueprints visual system didn't cover our implementation requirements. An example of that is when we created an Array

of structs to store value for the Q learning Implementation. Basically an int array that represents the rewards for the actions and its corresponding state name. Unreal's C++ is a bit different than the regular C++ syntax with added features that enable classes, functions, variables e.t.c to interact with the engine's interface and blueprint merging. An example is seen on figure 4.

```
19   USTRUCT(BlueprintType)
20   struct FStateAndActions
21   {
22       GENERATED_USTRUCT_BODY()
23
24       UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Basic)
25       TArray<int32> actionArray;
26
27       UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Basic)
28       FString stateName;
29   };
```

Fig 4. Example of C++ snippet in Unreal that describes a struct with an array and a string. Exposing a variable to the editor is done by using the special macro, UPROPERTY() . By passing more information to the macro we can call it in blueprints , put into categories and more. [13]

Considering that the implementation runs on a loop. the key points of the agent's algorithm can be described with pseudo-code as:

if CanSeeActor == false **then**
    Choose Location
    Move to Location
else
    while ActorsSensed != NULL **do**
        Add object to array ActorsSensed
        Calculate Score for object
        Move to Location of highest scored object
        Perform action (Interact || Shoot || Wait)
        Remove object from ActorsSensed
    end while
end if

"Choose a Location" on line 2 of the algorithm is based on equation (1).From this point on we will refer to the whole algorithm as Algorithm (1).

4.3 Q Learning

As discussed in the related work, since the level of repetition in performance testing is high, we wanted to prioritize the findings of performance issues for next runs and provide a way to give feedback faster. To achieve this we used reinforcement learning and more specifically Q-learning to reinforce certain behaviors that can lead to issues or bugs, based on previous results.

In Q learning , an agent interacts with its environment by perceiving its state and selecting an appropriate action, either from a learned policy or by random exploration of possible actions. As a result, the agent receives feedback in terms of rewards, which rate the performance of its previous action.

Moving towards our approach, the core idea is to map locations of the environment that can have big impact on the performance or map various actions that can lead to performance bugs / drops. To achieve that we instantiate certain milestones in key spaces of the environment. In detail, at the start of the game ,at the position of each obstacle (in our case "doors") a milestone is added on it's origin position. This is done automatically by getting the position of each obstacle with the help of the game engine functions. These milestones are programmed to produce a sound and can be perceived by the agent through its hearing sense. As a result the agent knows their locations from the very start. Their "Desire" score is also higher so they are prioritized over other objects. When the agent reaches a milestone a certain amount of time T is given before the agent moves to the next milestone. Within this specified time the agent is free to test as described on Algorithm (1) of section 4.2, meaning to interact with objects that he's seen and that are near him. If the performance is below a certain tolerance level the location is mapped. This is used to fill the reward matrix for the Q learning algorithm.

We map traversal between locations to the Q learning's reward matrix, such that Q(i,j) represents the reward of doing the action "go to location j" when the agent is in the location i. These locations in our case are the milestones. When a performance drop is mapped as seen above, a reward is given on the milestone according to how big the drop was. The bigger the drop, the better the reward.

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

More specifically, in the matrix above we consider the states as numbers, basically the number of the milestone and actions as go to state (milestone). Regarding the values inside the matrix, the 0 on the matrix indicates that there is a path between the two milestones but no reward is given when interactions happen nearby. The -1 indicates that there is no direct path between the two milestones. The 100 is the reward given if the milestone 5 (for the above example) is visited and a performance drop is noticed. If there is a -1 in the matrix and by interacting with an obstacle a path becomes feasible, for these two milestones the -1 becomes 0. This is how connectivity is mapped while creating the matrix. After the reward matrix is created we pass it on the Q learning algorithm.

The transition rule of Q learning is a very simple formula:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(r + \gamma max Q_t(s',a') - Q_t(s,a))$$

Here Q(s',a') denotes the action-state value of the next possible state, choosing optimal a (i.e. the next action), r the immediate reward provided by the reward matrix "R", and α is the learning rate of the agent. Having reached state s, action a must be chosen which maximizes, Q(s,a). This is referred to as greedy-method where in each state the best-rewarded action is chosen according to the stored Q-values. The term policy means assignment between states and actions. In each state, policy ϕ is defined as $\phi(s) = a$, where action a yields the maximal reward during the next step in state s. However, in some cases actions which do not assure good immediate rewards should also be chosen, because after a couple of decision steps the decision chain may provide much higher reward, than in the case of greedy steps. Therefore, the agent must carry out exploration when

it selects an action. We use the ϵ-greedy exploration method where the agents choose best action a with probability of 1-ϵ, and select alternative actions randomly with the remaining ϵ probability (ϵ $\epsilon[0, 1]$). This function gives a certain chance to avoid local optima and find maximal long run reward. Finally the γ parameter has a range of 0 to 1 ($0 <= \gamma <= 1$). If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward. Alpha notated as α is the learning rate has a range of 0 to 1 ($0 <= \alpha <= 1$). It determines to what extent the newly acquired information will override the old information.A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information [41] [28]

The pseudo code of Q-learning proceeds as follows:

(1) Initialize the Values of matrix Q(s, a) to zeros.
(2) Observe the current state 's'.
(3) Choose an action 'a' for that state based on one of the action selection policies (ϵ greedy)
(4) Observe the reward 'r' as well as the new state 's'.
(5) Update the Value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to the formula described above.
(6) Set the state to the new state Q_{t+1}, and repeat.

The Q matrix is converged in every step closer to the optimal solution. In the end it should look like this:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

We use the results of the Q matrix which our agent follows as described below to make him able to find the optimal paths to the performance drops.

We present certain algorithmic steps that map the most important parts of the retesting agent implementation:

(1) Create the Reward Matrix 'R' using the algorithm:

```
while Milestones left to visit != NULL do
    Move to a random unvisited milestone
    Set milestone to visited
    Start Countdown Timer with T=5 seconds
    while T < 5 do
        Use Algorithm (1)
        if Performance < Tolerance Value then
            Give reward to milestone
            Map Location
        end if
    end while
end while
```

(2) Run the Q Learning algorithm offline and get the Q Matrix.
    (a) We think of it as running a 2D simulation of rooms in which we converge the Q matrix step by step by choosing different states and actions as described above.
    (b) After many trials, the action-value function Q keeps improving and will converge to the optimal Q*.
(3) Use the converged Q Matrix and check which actions are choosed when traversing the Q matrix for each starting state.
    (a) The actions are choosed greedily by checking which actions have the highest Q value.
    (b) We also set two different termination criteria. The first one is when more than 20 steps happen while running the simulation starting from each state separately and the second one is if the maximum reward/ terminal state has been reached.
(4) Sum up the total number an action was choosed in the previous step and use the results in a form of prioritization for the agent to follow in the retesting run. For example with the mentioned matrices the endstate or else more desirable state would be state 5 and since when we reach state 5 for every traverse of different starting state, it is the most visited state and the biggest score is assigned to it.
(5) On the retesting run we use the pseudo-code of step 1 but instead of moving to random unvisited milestones they are prioritized as seen on the previous step.

Since our environment is pretty static we create the reward matrix online (while the agent plays the game) but create the Q matrix offline (not while playing). This could change in more complicated situations such as a dynamic environment but in our case it was not needed and saved as a lot of time. We propose this implementation as a basis for future work in that regard.

Note that the matrices shown above (Reward matrix and Q matrix) are not the real ones that are produced. They are shown for ease of understanding. A real example of the matrices can be seen on the appendix. Matrix example is taken from [28]

5 EXPERIMENT

To answer the research questions two experiment were conducted. In section 5.1 the game design and the props used are discussed. The experiment requirements and procedure are covered in section 5.2 and 5.3. The first experiment which is about normal performance testing, is covered in section 5.4 while the second one is about retesting and is covered in section 5.5.

Fig 5. Region in the experimental VR game

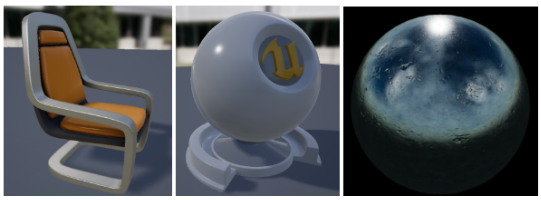

Fig 6. Starter Content: Example of free assets used, provided from Unreal Engine.

5.1 Design and props

For this study a VR FPS game was designed. To create the feel of a real game we added interactions with the environment and fighting with enemies. The space is divided in twenty rooms. Sometimes they are separated with obstacles, meaning doors that the player can open and gain access to them. An example of a region on our game can be seen on figure 5.

For the props the "Starter content package" that comes free with UE4 was used. It features a wide range of assets, everything from materials to particle effects , primitive shapes and various 3d objects. Example objects of the "Starter content package" can be seen on figure 6.

The games was made for mobile VR and can run on any android device that is above certain hardware requirements. A wireless game controller is also needed to control the character. For the experiment we used a Xioami Redmi Note 4 smartphone, a basic Google Cardboard and an OEM wireless controller.

5.2 Performance drop design

It's common knowledge that the graphic requirements of video games use most of the computation time. It is also verified by our results as you can see on the Figure 8, that rendering (RT , orange line) takes more time to be processed compared to the game thread (GT, blue line) or game logic. For the experiment there were several drops that can be evaluated. Some were created technically by us and some were already there due to implementation reasons.

List of Performance Errors:

(1) The first performance drop that is noticed is when the level is loaded. We didn't add a loading screen and everything is loaded at once when the game starts. This causes overhead but since it's a regular phenomenon it was not considered to be important and it seemed natural to most of the testers so we don't count it.
(2) The second performance drop was noted when an object is instantiated for the first time. We load everything in the beginning but the bullets that the player shoots are instantiated when the first shot is fired. This results in a slight performance drop which occurs most of the time but not always. This is a common problem in game development and may be solved by instantiating a number of bullets in a pool at the start of the game and recycling them. We didn't solve it since it added to our experiment.
(3) There are other performance drop that are not found 100% of the time. One of them is when a certain battle scene takes place with two enemies. The decisions and results when fighting can vary from run to run and sometimes can last longer and be more intense gameplay and rendering wise. This can cause performance drops.
(4) Five performance drops were technically added on five different rooms. We used and invisible trigger box that when the player overlapped with it an explosion of particles happened, causing the performance to drop for a couple of seconds. Three of them were easy to find, as they were inside regions that required from the player to enter to be able to proceed to the game. Two of them were added in "dead" regions that the player would not normally visit and doesn't contain something for the agent to perceive and find interest to move to. One was even harder to find than the other as seen from our experiment, at least for the agent.
(5) Performance drops can also happen due to gameplay errors. In our game we had fifteen items that the player can collect. In case the player shot at those items, a performance drop was triggered. We added this specific bug to simulate problems that can happen during the game development cycle and require from the testers to think outside the box in order to find them. We also try to note the importance of not playing in a fashion that the designers want to.
(6) Performance spikes can happen at random and not all are controllable. There are various issues for that. When garbage collection happens for example, there might be a small drop in performance. The garbage collector is an important part of the engine and will automatically delete objects when they are no longer needed. An object is no longer needed when it is no longer referenced by any other object. [15]
(7) There was a performance issue that occurred only in a specific device of one of the testers. We also asked the participant to use the agent and identify if the same data can be collected.

To conclude we had 20 performance drops available that we knew about , happened with 100% reproduction rate and expected the agent and the testers to find. We consider these the main findings while everything as additional findings.

One of the problems we had to solve is how the agent will report these problems back once they were found. When a drop happened we used the in built function of the game engine to capture a screenshot and save it. We also saved the global location (XYZ) of the player and exported it into a file. With these information a developer could use them to reproduce the error quite easily. This works well for the bugs that happen without interacting with the environment , but if an interaction is required for the bug to trigger these are not enough because you likely wouldn't understand the interaction needed to reproduce the drop. A video could be captured in the entire playthrough as a solution so you can see the full interactions needed but we didn't go to that extent.

5.3 Methods and experiment procedure

We performed two different experiments. For the first one we tested how our agent performs versus manual testers. We hired five professional testers, gave them detailed instructions about the world and asked them to report any performance drops found during their playthrough. We also got the detailed performance log that is auto generated on the device after a completed playthrough to compare

the general performance. Last, the time required for the manual tester versus the agent's time required is evaluated.

For the second experiment, we evaluated the time required for the agent to retest the most critical drops versus one manual tester that was already experienced in the game and was given instructions of which regions to retest.

5.4 Performance drop finding experiment.

The experiment for the manual tester part consists of the following parts:

(1) **Consent Form.** To inform the participant about experiment and warn him/her about the risks of VR.
(2) **Environment familiarization.** The participants are shown the regions of the game via pictures, so they can familiarize with it.
(3) **Game and testing Instructions.** The participants are informed about the controls of the game and their objectives.
(4) **Testing the VR game.** The participants are testing the game and mark any performance drop.
(5) **General and reporting questionnaire.** The participants answer some general questions and report any issues found about the game.

The experiment for the agent consists of the following parts:

(1) **Agent play mode** We run the agent ten times to collect data with the purpose of playing the game like a human would, or more like the designer's expectations of playing the game are. We recorded the performance drops found and also the time required for the completion of the game.
(2) **Agent testing mode** We run the agent ten times with a purpose of testing the game, meaning not finishing the game but wondering around until all performance drops are found and doing unorthodox actions that can lead to bad behavior of the game. We did that by tweaking the knowledge base. For this experiment we focused only on the main findings (errors (4) and (5)).

5.4.1 *Results for Performance drop finding experiment.*

Agent playing the VR game

Let's start with the agent performance while playing the game. The agent could complete the game with an average time of 4 minutes and 55 seconds. In the sum of the runs he found all the performance errors that he was capable of finding. The only errors that were out of its capabilities were the ones mentioned as gameplay errors (5). By tweaking the knowledge base to make the agent able to shoot at items he became able to find them but that is not considered as required behavior to complete the game.

Manual testers playing the VR game

The manual testers played the game with a purpose of finding performance drops. Almost all the performance drops were found (no matter hard or easy). Also 4 out of 5 found the gameplay errors meaning they did try to shoot at the statues while the instructions of the game that were given stated otherwise. In terms of time requirements , when the tester was not familiar with the game it took them around 20 minutes to find all the performance drops.

Agent playing mode results		
Performance drop caused by	**Found**	**Number of runs in which they were found**
Easy to find rendering errors (3 out of 3 were found in every case)	Yes	10/10
Hard to find rendering error	Yes	5/10
Hardest to find rendering error	Yes	1/10
Gameplay errors (15 out of 15 were found in every case)	Yes	0/10
Instantiating error	Yes	7*/10
Intense gameplay fight scenes error	Yes	3*/10
Random	Yes	5*/10

Manual testers results		
Performance drop caused by	**Found**	**Number of runs in which they were found**
Easy to find rendering errors (3 out of 3 were found in every case)	Yes	5/5
Hard to find rendering error	Yes	4/5
Hardest to find rendering error	Yes	5/5
Gameplay errors (15 out of 15 were found in every case)	Yes	4/5
Instantiating error	Yes	3*/5
Intense gameplay fight scenes error	Yes	1*/5
Random	Yes	4*/5

When the tester became familiar with the environment and the game, time requirements drop dramatically to around 5 minutes.

Agent testing mode

The results for agent when he was in testing mode proved to be quite chaotic in terms of timing. He had though 100% success of finding all the performance drops.

Agent testing mode results		
Performance drop caused by	**Found**	**Number of runs in which they were found**
Easy to find rendering errors (3 out of 3 were found in every case)	Yes	5/5
Hard to find rendering error	Yes	5/5
Hardest to find rendering error	Yes	5/5
Gameplay errors (15 out of 15 were found in every case)	Yes	Based on Knowledge Base
Instantiating error	Yes	4*/5
Intense gameplay fight scenes error	Yes	2*/5
Random	Yes	5*/5

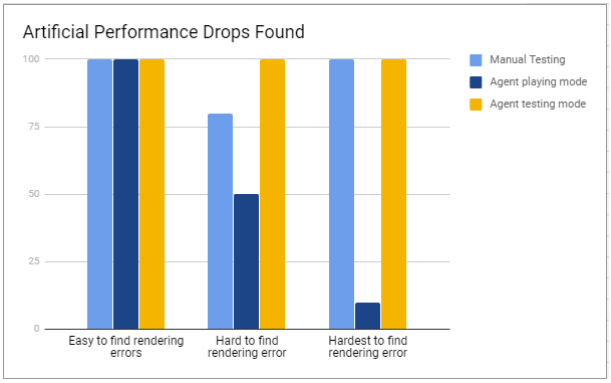

Fig 7. Results for artificially created errors. On the vertical axis we can see the percentage of errors found for each corresponding category.

One problem with this method was the chaotic time consistency which ranged from 4 minutes and 36 seconds to 25 minutes and 12 seconds. The Agent is less consistent in time than the manual tester due to the randomness in the pattern of moving when no items are perceived. This can mean that in bigger environments, it could be more time depending. Last about the gameplay errors we have to note that we tweaked the values on the knowledge base in some runs to simulate "out of the box" behavior. This enabled the agent to find the gameplay errors. Of course not all out of the box behavior can be programmed so this is not the most optimal solution.

*Asterisks in the tables mean that errors didn't happen in every run. The agent was able to identify them every time they happened though. For example Instantiating error doesn't happen 100% of the time but when it happens it is reported.

5.4.2 *Special case testing.*

One of the testers reported he had a field of view (FOV) fluctuation issue that made it hard for him to complete the test. Basically huge drops were happening when the field of view changed rapidly. For example when he was looking at a wall and quickly changing his orientation resulting on looking at the opposite direction.

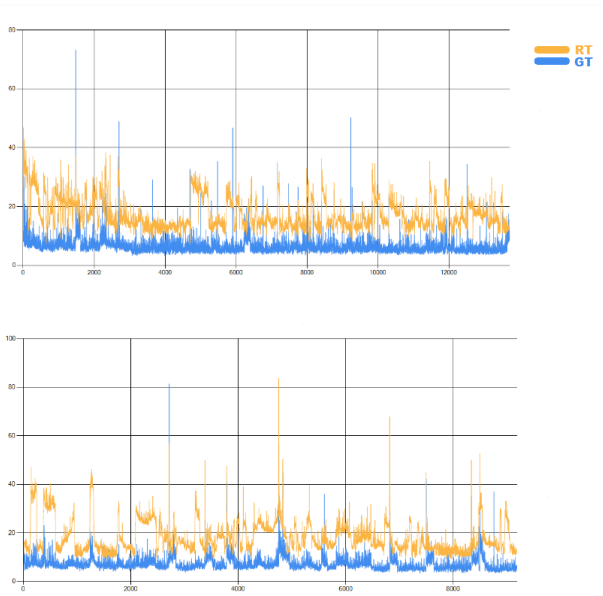

Fig 8. Graph comparison for Special Case Testing between Agent (above) and Manual (below). Horizontal axis x shows the number of the frames and vertical axis y shows the computational time required for the certain frame to processed in milliseconds (ms). Orange line represents the Rendering thread (RT) and blue line represents the game thread (GT)

We wanted to see if our agent would recognize the problem. We asked the tester to run the automated agent on his device and observe the performance drops. He told us that the problem occurred and was exactly the same. This shows that our agent would provide good results even in situations that happen in a real development environment.

We can see from the two Frame Rate graphs above that the patterns that they follow are similar. By similar we mean the fluctuations that are observed on the orange line/ rendering thread (spikes happen when performance drops occur). Furthermore the average FPS recorded in this particular case by the user playing was **48.73** while the average FPS recorded by the agent was almost the same **52.29**

5.4.3 *Discussion.*

As the results indicate from the tables and figure 7, the agent can perform good in terms of finding performance errors that are based on graphic issues and gameplay actions that are scripted by the programmer. This answers our first research question (RQ1) by showing that it is possible to automate performance testing in VR worlds with 100% success rate, at least for certain parts.

We can't really support though, that the agent will be able to find performance drops caused by functional errors or player actions that are not anticipated. Having a knowledge base that can differentiate in terms of gameplay actions can definitely help; however not everything can be predicted. This is the major cause that forces us to support the claim, that agents can't replace human testers as of now. Resulting in answering the RQ1 on a more negative way but not absolute. It also helps us answering RQA by showing the potential of increased intelligence in agents. It might be true that even an agent with very limited to no intelligence at all could provide

some results about performance testing. However by seeing that more complicated issues can cause performance errors we believe that the more intelligent the agent the better the findings will be. This problem can be considered as a requirement engineering problem that balances work needed to increase the agent's intelligence versus using manual users to do the testing.

Next, the reporting of the problems can't be as accurate as one of a human user. While communicating with the testers I understood that additional information can be provided by them that can be essential and really difficult to collect with the agent. An example of that could be the performance fluctuation drops mentioned on 5.4.2. While communicating with the tester he provided an accurate description of the problem that an outside observer would have difficulties understanding by looking at the currently generated reports. Even if a video captured the whole gameplay action it might be difficult to tell what the real problem is just by observing it.

On the other hand we can never be sure that the human tester will be 100% accurate in his reports or if he will even notice that the performance errors always happened. For example in an intense fighting scene the tester might not notice that the performance dropped below the acceptable level because he is too focused on the actions that he needs to do. In contrast to that because the agent is absolute in terms of numerical values (e.g. when the framerate for the last 20 consecutive frames is below threshold, save the location). As a result we can be sure that no performance drops will be missed. The threshold should be a bit lenient and not absolute e.g 60 frames per second. A small spike to 40 for example in one frame may not even be noticeable. Instead a certain number of frames can be checked together and if the drop is constant and below a specific threshold, it is then reported as a performance error.

It is also important to note that there may be false positives; these are performance drops reported by the agent, but are not actually noticeable by the users. In our case once the appropriate tolerance level was found the false positives were minimal; For our agent experiment while in playing mode, only 2 false positives were noticed during 10 play tests.

Finally, we notice that an inexperienced user requires as much time to cover the testing procedure as the agent in testing mode. However it can be reduced dramatically when the user gains experience with the environment, resulting in overall better productivity. We also had our agent to learn from previous experiences, so we created as discussed above a method of mapping crucial test cases on the environment that the agent can learn to navigate to and as a result find performance errors faster. In the next section we discuss the experiment based on this method.

5.5 Retesting experiment

The experiment for the manual tester part consists of the following parts:

(1) **Consent Form.** To inform the participant about experiment and warn him/her about the risks of VR.
(2) **Game familiarization.** The user was part of the first experiment and was given extra time to familiarize with the game even more.
(3) **Test cases instructions.** The participant is informed about the locations and/ or actions they need to prioritize. Specifically pictures of the rooms were given that the tester was able to recognize due to his experience with the environment.
(4) **Testing the VR game.** The participants are visiting the testing locations and mark any performance drop while timing their playthrough.

The experiment for the agent part consists of the following parts:

(1) Mapping performance drop locations in a reward matrix that connects them.
(2) Running Q learning algorithm to find optimal traversal route.
(3) Applying the results and record time required to retest certain locations first.

5.5.1 Results for Retesting experiment.

Manual Testing.

The results for the manual tester are pretty straightforward. The tester was able to recognize the locations and move to them in the fastest way possible. The average time required to retest all locations was **1 minute and 29 seconds** with very small deviations.

Q Learning based agent.

The average time required to retest all locations was on par and even slightly better than than the manual testing experiment with average time **1 minute and 24 seconds**. This results are after all the learning procedure is completed. The milestones that the agent prioritized were the best possible options, meaning that the agent , if there was a path available, moved on the region that had the lowest noticed performance first. One last mention is that while the agent retested all the main findings he also retested all regions that were mapped while creating the reward matrix and are seen on the next table marked as additional findings . That doesn't mean thought that he always re-found the error in the additional findings regions since they don't happen with 100% reproduction rate.

5.5.2 Discussion.

The results answer our RQ2 about retesting. We implemented a method that is on par with human testers in terms of time required to retest. Nevertheless there are still some things that can be improved.

As we have seen above the learning procedure uses a free roaming procedure like the testing mode , a run that creates the reward matrix and running the q learning algorithm to produce the Q matrix. The time required for everything can deviate greatly since it requires the agent to run in testing mode. The whole procedure can take a lot of time especially in larger and complicated environments but the benefits can be great as we see in our results. Unfortunately in case all the performance drops locations change everything should be re-learned from the start.

On the contrary while the tester should also be experienced with the virtual world, a large amount of that experience remains and even if the performance drop locations or if the design of the world changes a bit, there shouldn't be a problem for a manual tester to cope with it. This probably puts the manual tester ahead of the game again.

Another possible problem of our approach is that on testing mode, locations of random drops are also mapped and they are re tested along with the performance drops that happen 100% of

Retesting Agent timings	
Additional Findings	**Time**
3	1.19
3	1.27
1	1.27
2	1.14
5	1.34

the time. This may require extra time requirements but in our case of limited space it didn't seem to affect the results that much. As we can see on the table above (Retesting Agent timings) the resulting additional findings doesn't seem to matter that much for our occasion and the time required remains pretty much the same.

6 CONCLUSION AND FUTURE WORK

In case of a larger environment the agent would probably require a lot of time to traverse from one place to another with all the required gameplay actions in between. The required sequence of actions needed in this case could be very long. What we would like to do then, is minimize the sequence to obtain a new test sequence that reproduces the error as fast as possible. There has been substantial related work in traditional software testing for this kind of problem. One of them is "Guided Algebraic Specification Mining for Failure Simplification" which addresses the problem of log reduction by rewriting the reported log in such a way that it preserves the ability to reproduce the same error[11]. The paper presents an algorithm for rewrite rules inference, and a terminating reduction strategy based on inferred from a set of predefined algebraic rewrite rule patterns that transform the log reports into a Finite State Machine (FSM) procedure. A log is also generated in games (at least with UE4) which provides ways to track actions and errors and the FSM closely resembles the behavior tree system we use. Another solution for failure simplification is Delta debugging (DD). In a few words, DD tries to replay various subsequences of the original failing sequence, starting from binary splitting and gradually increasing the granularity until a smaller subsequence is found. Depending on the size of the virtual world this could potentially take huge amounts of time. Nevertheless various researchers have studied on how to minimize the time required for DD as we see in the paper "Reduce First, debug later"[12] where by combining the approach of log reduction seen on "Guided Algebraic Specification Mining for Failure Simplification"[11] they proved an improvement of the DD execution time.

Next, we want to note that our solution of retesting would need a lot of tweaks and adjustments in case that is applied in a different and more complicated virtual world. In terms of a world that is more dynamic the Q matrix should be converged online to minimize any possible differentiations. This would also show the positives of this implementation over solving a path planning network with a static algorithm like Dijkstra. Dijkstra's algorithm is a method for finding the shortest paths between nodes in a graph, which could easily represent our milestone network and solve it in a faster and easier way [50]. However it can't cope with dynamic environments as we have seen in the related work section where we discuss the paper "Reinforcement learning for solving shortest - path and dynamic scheduling problems"[41]. Their method demonstrates the advantages of the RL implementation in dynamic environments. They do that by introducing a value named "Temperature" which greatly affects the learning behaviour. When this is high the learning capabilities also become high and thus fluctuations in the delivery time are observed. The goal to this is to reduce the temperature of the system in order to reach optimal behaviour. In a basic sense it can be considered that an increase in temperature when delivery of packages is slow , should increase learning capabilities of the agent to converge into more optimal values and finally when this is achieved , the temperature value to be decreased and follow the optimal policy. In the same way when testing of certain regions takes long times or is unfeasible we could re evaluate the learned value function or policy accordingly. This can also work to evaluate dynamic conditions that may make the testing not feasible at all. For example what if there is an enemy that must be beaten in order to reach the next milestone? And what if there is a certain weapon that must be obtained in order to beat this enemy? Increasing the scalability of the experiments and tweak the Q algorithm to provide results for such a case would be an interesting direction for future implementations.

To conclude we successfully showed that testing of performance errors focused on heavy rendering for VR can be automated and possibly unload a huge amount of workload from manual testers if used right. We also showed that the agent can recognize real world problems and not only artificially made as shown on the chapter 5.4.2 Special case testing. It remains yet to see if the ideas presented in this paper can work in a modular fashion and be incorporated in a modern game development environment.

We also provided a way to do retesting and learn from experience so we can have better timings when a rerun to test specific problems is performed. The implementation provided results on par with human testers in terms of timing. Of course substantial work of has to be done to work in more complicated and dynamic environments as mentioned above.

Nevertheless the experiments are promising and if the core implementation of the agent can expand and work with minor adjustments in any first person VR game, it would prove a valuable asset to any company. It could even expand in a more adaptive way for regression testing, meaning to also check other potential problems that could occur based on if a fix was found.

Finally we have to note that the next big step in the research is to implement this method in a commercial VR game , research the requirements needed to do that and finally find out what's the impact that can be made. This will provide more integrity to the research approach and ideas behind it and possibly convert into a fully working testing framework.

A APPENDIX

The contents of the appendix proceeds as follows. On section A1 we provide some extra information about Virtual Reality. Next, on section A2 we provide some more info about testing in games. Moving on section A3 extra related work is provided that is connected to our research. The experiment details and procedure are described on section A4. Last on section A5 we show some more details of our research approach and the appendix is concluded.

A.1 Virtual Reality Information

As already discussed companies like Samsung and Google made VR possible in the mobile market by introducing head mounted displays (HMD) that you can slide your smartphone into and provide a VR experience. The smartphone acts as the headset's display and processor, while the HMD unit itself acts as the controller, which contains the field of view, as well as a custom inertial measurement unit, or IMU, for rotational tracking, which connects to the smartphone via micro-USB. Google introduced cardboard. Named for its fold-out cardboard viewer, the platform is intended as a low-cost system to encourage interest and development in VR applications. Users can either build their own viewer from simple, low-cost components using specifications published by Google, or purchase a pre-manufactured one. To use the platform, users run Cardboard-compatible applications on their phone, place the phone into the back of the viewer, and view content through the lenses. Samsung Introduced Gear VR which is a more comfortable headset that resembles traditional HMD's like Oculus with adjustable head straps and options of focuses to whatever your eyes need. It also has its own marketplace with seperate apps.[39] [49]

We decided to implement a First Person Shooter game for the reason of almost 75% of VR games are being controlled from a first person perspective. This number is based on the database of the web site "VR Games for"[17] which features an impressive list of published and upcoming Virtual Reality games for the most known platforms. For each game they provide information such as genre, platform, operating system and various features that describe it gameplay wise. Since the list was quite extensive in size, with over 2400 titles we used the software Selenium [52] to collect the required data. It is a portable software-testing framework for web applications. Selenium provides a playback (formerly also recording) tool for authoring tests without the need to learn a test scripting language (Selenium IDE). We used it to navigate through the list and collect the genre of its game. Then we calculated the results.

A.2 Testing on games

First thing first, it is noteworthy to mention the two basic classes of software testing, namely black box testing and white box testing. Black box testing is the testing process that ignores the internal mechanism of a system or component and focuses solely on the output generated in response to selected inputs and execution conditions. White box is testing by taking into account the internal mechanism of a system or component.[22]

A.2.1 Testing steps.

Testing plays a very important role in Games. A game is tested at different level of its development process. Most of game testing is black box testing. Developers don't really test their own games, neither have time to , nor is it a good idea to test by themselves. Quoting the authors of "An Overview of Game Testing Techniques"[6] game testing is performed in the following six step order:

- **Plan and design the test:** Although much of this plan is done earlier in the software test plan document, with every new prototype of the game, this documents need to be revisited to update any change in the specifications, new test cases, new configuration support. The tester should insure that no new issues were introduced.
- **Prepare the test:** All the teams should update their code, tests, documents and test environment and align it with one another. The test development team should mark the bugs fixed, and the test time should verify them.
- **Perform the test:** Run the test suit again. If any defect is found, test around the defect to make sure that the bug is verified.
- **Report the results:** Complete details about the bugs are reported.
- **Repair the Bug:** The test team participates in this step by explaining the bug to the developing team and provides direct testing to track the bug.
- **Return to step 1 and retest:** A new build is produced after one cycle.

A.2.2 Types of testing.

Among others, black box testing includes functional testing, stress testing, performance testing and usability testing. According to the authors of "Testing overview and Black Boxing Techniques"[56] :

Functional testing involves ensuring that the functionality specified in the requirement specification works. For example this could be a specific action of the game e.g pressing the A button shoots a bullet.

Compliance or system testing involves putting the game in many different environments to ensure that it works in typical user environments with various versions and types of operating systems and /or devices and specific parts of devices.

Stress testing is conducted to evaluate a system or component at or beyond the limits of its specification or requirement. For example this could be over using a certain action to see if the results break the application e.g. shooting at a destructible object too many times.

Usability testing is conducted to evaluate the extent to which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. While stress testing can be and is often automated, usability testing is done by human-computer interaction specialists that observe humans interacting with the system. In VR this holds more applications as it is required that no motion sickness is occured while playing the game. On the other hand a tester gains resistance to motion sickness as he is continuously presented to it. As mentioned on best practices of Occulus[45] users with no experience in VR should test the applications to get appropriate results as they provide better samples and are more objective towards it.

Performance testing is testing conducted to evaluate the compliance of a system or component with specified performance requirements . When the frame rate drops below the required levels,

performance errors are present and a variety of reasons can be the cause. It should also be mentioned that performance bugs can cause excess motion sickness in VR and must be avoided at all costs. [56]

A.2.3 Definition of errors in video games.

One important mention of literature in game testing is the work "What Went Wrong: A Taxonomy of Video Game Bugs" [27] where they present a taxonomy of possible failures that can happen in video games. They support that a categorization of buggy gameplay experiences is possible and can lead to new methods of analyzing and solving video game bugs. Quoting the authors, some of them are:

Object out of bounds: Object out of bounds for any state 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 graphical representation: An Invalid graphical representation occurs when a certain aspect of the world state is being rendered incorrectly.

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 is another broad category that catches bugs related to an NPC performing some act that breaks 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. This category includes seeing through walls or gaining complete information on a game map that should have a fog of war.

Invalid position over time: This category describes invalid movements, such as rapid accelerations or hovering in the air, but can also include objects teleporting around the world due to poor physics or faulty world updates. This category also describes the lack of expected movement.

Invalid context state over time: Invalid context state over time applies to objects that stay in a state for too long or too infrequently. State is used only to mean the user-observable characteristics that an object is showing, not the actual fags used in the implementation.

Implementation response issues: Implementation response is the category most closely aligned to how the game interacts with the base hardware. This category covers failures where some aspect of the hardware is not performing at an optimal speed. Such failures include network lag, input lag (time from pressing a button to something occurring in game) or frame rate fluctuations.

All these types of errors can potentially lead to performance errors. We also need to note that if any of these errors happen our agent wouldn't possibly respond in the best possible way or maybe even understand that something is wrong. This is another problem of having agents test the game instead of manual testers. The manual testers would report any other issues found along the way while the agent will either crash or ignore the errors. Our implementation focuses mostly on the end of development cycle where the rest of the errors are minimized.

A.3 Extra Related work

In this section additional related work will be provided about the state of the art technologies used for agents in video games and works about testing in regular software.

A.3.1 Path Planning.

In video games, every entity that can move and describes some form of intelligence must plan their motions in the virtual world. We will take a look at two of the most popular approaches used in modern video games.

A.3.2 A planning.*

First we will take a look at the survey; Path planning: A 2013 survey [33]. The authors presented literature review on path planning methods. This survey confirms the most common belief about path planning in video games, meaning that the A* algorithm is the undoubtedly victor in terms of popularity for video game development. Before going deeper into A* let's take a look at the grids.

Quoting the authors, regular grids are the most used in environment modeling especially in robotics and video games. They have several advantages such as the ease of implementation and the simplicity of updates. Because regular grids always have the same number of nodes and edges regardless the number of obstacles that may exist. Three types of cells are used in the state-of the-art which are Square, Triangular and Hexagonal grids. In figure 1 we can see an example square grid.

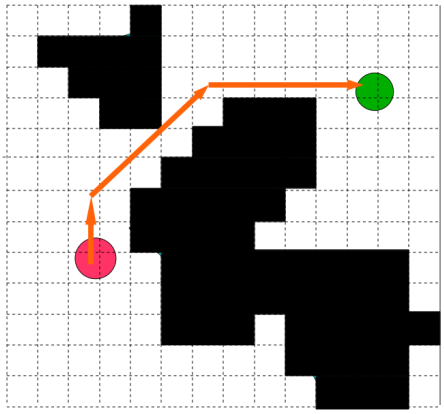

Fig 1. Square grid example with obstacles (figure taken from [24])

A*, pronounced as "A star", is the most well-known pathfinding algorithms . The A* algorithm g offers an acceptable solution when applied with a grid. It can be described as solving a graph node, by setting the initial position as a (node) and place it on a list named "Open", along with its estimated cost to the destination, which is determined by a heuristic. The heuristic is often just the geometric distance between two nodes (Euclidean and Manhattan distance are the most common ones). Then perform the following loop while the Open list is nonempty:

(1) Switch the node that has the lowest estimated cost to the destination to another list named "Closed List".

(2) If the node is the destination, we've successfully finished (quit).
(3) Else examine the node's eight neighboring nodes.
(4) For each of the nodes which are not blocked, calculate the estimated cost to the goal of the path that goes through that node. (This is the actual cost to reach that node from the origin, plus the heuristic cost to the destination.)
(5) Push all those nonblocked surrounding nodes onto the Open list, and repeat loop.

The algorithm stops if the target goal is not found (all nodes are in the closed list) or if the goal is reached .There are various improvements over the A* algorithm like D* which is dynamic and react quickly when there is a change on the studied map or the Field D* which does not constrain movement to a grid, instead the best path can have the unit moving along any angle and not just 45- (or 90-) degrees between grid-points [33]. Since our main focus is not path planning we will not go into further detail about A* and accompanied implementations for this thesis.

A.3.3 Navigation meshes.
One of the most interesting mentions of the above survey is the navigation meshes (navmesh) [33]. It is a common strategy for efficiently computing realistic paths with partition the environment into a collection of walkable areas. This method is growing in popularity as game engines such as Unity and Unreal have implemented it internally.

Navigation mesh is a collection of two-dimensional convex polygons that define which areas of an environment are traversable by agents. In other words, a character in a game could freely walk around within these areas unobstructed by trees, lava, or other barriers that are part of the environment. Adjacent polygons are connected to each other in a graph. Pathfinding within one of these polygons can be done trivially in a straight line because the polygon is convex and traversable. Pathfinding between polygons in the mesh can be done with one of the large number of graph search algorithms, such as A*. Agents on a navmesh can thus avoid computationally expensive collision detection checks with obstacles that are part of the environment. [33] [51]

We take a closer look with the literature Navigation Meshes for Realistic Multi-Layered Environments [43]. The authors model a complicated environment and use multi layered navigation meshes. A multi-layered environment is represented by a set of two dimensional layers and a set of connections. Each layer is a collection of two-dimensional polygons that all lie in a single plane, and each connection provides a means of moving between layers.

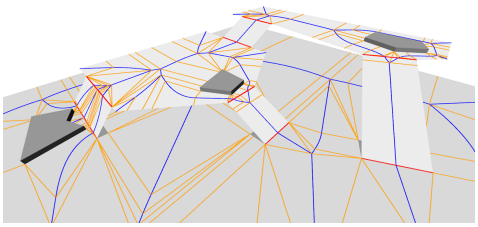

Fig 2. Example navigation mesh, multi layered (figure taken from [43])

They first compute the traditional medial axis of each two dimensional layer in the environment. The medial axis of these layers is the set of all points having more than one closest point on the object's boundary. The connections are then used to iteratively merge this collection of medial axes into a single data structure. By adding a linear number of line segments to this structure. A navigation mesh is obtained that mathematically describes the walkable areas in a multilayered environment. An example of the medial actions connections and a complete multilayered environment with navmeshes can be seen on figure 2.

Unreal Engine has its own path planning method with navmeshes integrated. We find it pretty much satisfactory and it covered our needs without problems.

A.3.4 Real Time path planning. Last mention about path planning is the paper Real Time Path Planning in heterogeneous environments [30]. This work is focused on character preferences for the traversable region types. They take into account various actions that occur in real life, for example pedestrians may prefer to walk on sidewalks, but they may occasionally need to traverse roads and dirt paths. By contrast, wild animals might try to stay in forest areas, but they are able to leave their protective environment when necessary.

To achieve that, they use the indicative route, which is a curve that passes through traversable region that a character wishes to use. But this path is not absolute. The character also steers to avoid collisions. For the purpose, a number of attractions points are computed. They are like milestones before reaching the last goal. A character chooses between different attraction points based on weights. Attraction points use reference points that are the closest to the character position between an attraction point and the previous reference point (First reference point is the initial character position) and two different distances. How far can attraction points be and how far is the reference point from the farthest attraction point.

This provided as an inspiration for the way we mapped the environment with the milestones and their corresponding score. This method could also prove as an alternative in case we need to remove the learning part from the agent implementation for a more stable solution.

A.3.5 Movement in VR. Movement for the player avatar in VR games differs from traditional video games. This is because movement and acceleration most commonly come from the user's avatar moving through the virtual environment (by locomotion or riding a vehicle) while the user's real-world body is stationary[45]. These situations can be discomforting because the user's vision tells them they are moving through space, but their bodily senses (vestibular sense and proprioception) say the opposite. This illusory perception of self-motion from vision alone has been termed vection, and is a major underlying cause of simulator sickness [25].

As such VR developers created the "teleport" like movement. By pointing somewhere in the environment the user can teleport there instead of the actual player pawn moving to the goal. This method is accumulated by popular AAA titles like the upcoming DOOM VR and Robo Recall [46].

We didn't use this method for moving our agent but even if we did, it shouldn't change the results. If this method becomes the standard of moving in VR though , it should be incorporated accordingly.

A.3.6 Behavior trees. Following from the introduction of behaviour trees in the paper, we take a more extensive look at some functions and procedures that they follow.

There are many different implementations of behavior trees. Here we will analyze some of the key concepts that define them.

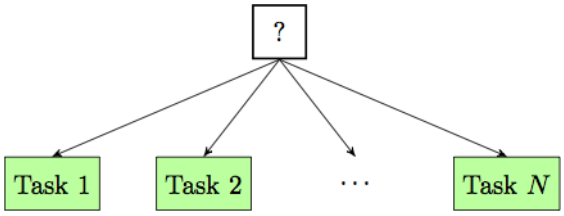

Fig 3. Selector Node with N tasks (figure taken from [48])

A.3.7 Key concepts of BT's. The execution of a BT starts from the root which sends ticks with a certain frequency to its children. A tick is an enabling signal that allows the execution of a child. When the execution of a node in the BT is allowed, it returns to the parent a status "running" if its execution has not finished yet, "success" if it has achieved its goal, or "failure" otherwise. A particular node or branch in the tree may take many ticks to complete. In the basic implementation of behavior trees, the system will traverse down from the root of the tree every single frame, testing each node down the tree to see which is active, rechecking any nodes along the way, until it reaches the currently active node to tick it again. When this happens , it will again have the opportunity to succeed, fail or continue running. [48] [5]

A control flow node is used to control the subtasks of which it is composed. A control flow node may be either a selector (fallback) node or a sequence node. A graphical representation of the selector can be seen on figure 3.

The selector node executes all the children in succession until it finds one that doesn't fail. Pseudo of the two control flow nodes follow taken from [48]

```
Pseudocode:
1 for i from 1 to n do
2     childstatus ← Tick(child(i))
3     if childstatus = running
4         return running
5     else if childstatus = success
6         return success
7 end
8 return failure
```

The sequence node runs until it finds a child that fails.

```
Pseudocode:
1 for i from 1 to n do
2     childstatus ← Tick(child(i))
3     if childstatus = running
4         return running
5     else if childstatus = failure
6         return failure
7 end
8 return success
```

This is more or less the key concepts in general but since Unreal Engine 4 (UE4) will be used for the project we will take a closer look at how behavior trees are treated there.

A.3.8 Unreal Engine Behavior trees. The implementation of BT's in UE4 follows an event driven method. It is based on event-driven programming , a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Event-driven behavior trees avoid doing lots of work every frame (like we've seen above with the constant ticking per time for checking the whole tree). Instead of constantly checking whether any relevant change has occurred, the behavior trees just passively listen for events which can trigger changes in the tree. This makes it more optimized and easier for debugging since you have knowledge for example about when and where a control flow node failed. To achieve the event driven behavior they make an important change on the architecture. The control flow nodes are not task leaf nodes that just succeed or fail but are used with decorators instead. Decorators have exactly one child leaf node. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node. So instead of ticking the entire tree again they tick that specific child directly. An example can be seen on figure 4 where the decorator is named CloseEnough and checks if someone is close enough to the agent. If this is true then the sequence is activated.

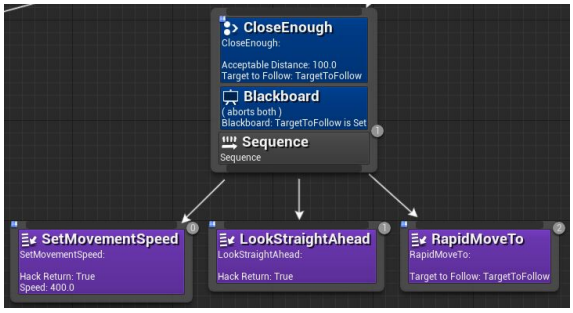

Fig 4. Decorator example in Unreal Engine [14]

Concurrent behaviors are also treated differently than normal. Standard behavior trees often use a Parallel composite node to handle concurrent behaviors. The Parallel node begins execution on all of its children simultaneously. Special rules determine how to act if one or more of those child trees finish (depending on the desired behavior). For optimization , simplicity and ease of use

Unreal uses simple parallel nodes. Simple Parallel nodes allow only two children: one which must be a single task node (with optional decorators), and the other of which can be a complete subtree. You can think of the Simple Parallel node as "While doing A, do B as well". For example, "While attacking the enemy, move toward the enemy." Basically, A is a primary task, and B is a secondary or filler task while waiting for A to complete. While there are some options as to how to handle the secondary meanwhile task (Task B), the node is relatively simple in concept compared to traditional Parallel nodes. Nonetheless, it supports much of the most common usage of Parallel nodes. Simple Parallel nodes allow easy usage of some of the event-driven optimizations. Full Parallel nodes would be much more complex to optimize.[9]

A.3.9 Learning AI in games. The majority of current approaches for game AI lead to predefined, static and predictable game agent responses, with no ability to adjust during game-play to the behavior or playing style of the player. Machine learning techniques provide a way to improve the behavioral dynamics of computer controlled game agents by facilitating the automated generation and selection of behaviors, thus enhancing the capabilities of digital game artificial intelligence and providing the opportunity to create more engaging and entertaining game-play experiences. In this literature review we will focus on three of the most used methods for machine learning in video games. Neural Networks, evolution and reinforcement learning. [26]

A.3.10 Evolutionary Machine Learning. Evolutionary computation comprises a class of optimization techniques which utilize simulated evolutionary processes in order to search through a problem space for an optimal solution. By representing the solution to a problem in genetic terms, a population of potential solutions is generated and maintained through the use of genetic operators. Each potential solution is encoded as a genotype, represented by a chromosome string, which is decoded into its corresponding phenotype form in order to enable the fitness of the genotype to be evaluated. Evolutionary methods enable the process of learning to be considered as a special case of optimization, whereby the learning process attempts to discover an optimal solution based on the fitness of individuals within a population of solutions [21] As a starting point of our learning literature we look at the work Evolving Behavior Trees for the Commercial Game DEFCON [4]. Their goal is to to highlight the potential for evolving behavior trees as a practical approach to developing AI-bots in games. They achieve it by designing and developing an AI-controlled player for the commercial real-time strategy game DEFCON. In particular, they evolved behavior trees to develop a competitive player which was able to outperform the game's original AI-bot more than 50 % of the time. At first they implemented a method to create randomly generated BT's that have different values for various actions of the game (e.g where to place troops, buildings..). Then they apply genetic operations by using crossovers on branches and mutations on nodes. An AI-bot was constructed with a controller that used the best trees evolved for four different in game behaviors. To define the best result available they used fitness functions. The fitness function simply defined is a function which takes a candidate solution to the problem as input and produces as output how "fit /good" the solution is with respect to the problem in consideration. The main problem with genetic methods to work, is that they need many generations of a lot of individuals to provide good results and the fitness functions take immense time for the processing. Quoting the authors of the mentioned paper; With each game taking approximately 90 seconds to complete, a total time of 3.6 million seconds (41 days) of continuous processing would be required for the project.

We can easily understand that using advanced AI using evolutionary behavior trees in a similar way, as it stands now may be unsuitable for testing of large scale games due to time requirements. We didn't use this approach since we don't consider that having an agent who is more competitive would help in performance testing. Simple BT's can work just fine and if for example we need better fighting capabilities in order to beat a certain enemy we can always "cheat" by adding more health or stronger bullets without possibly affecting performance testing.

A.3.11 Neural Networks. Neural networks (NN) present a class of learning models that are capable of providing a robust approach to learning discrete-valued, real-valued or vector-valued target functions. They can consist of a number of possible network topologies, incorporating varieties of different architectures containing a choice of feed-forward, feedback and lateral weighted connections between neurons, neural network learning has been applied to a range of learning tasks. [29] To get a bit more familiar with NN let's take a closer look at Integrated Machine Learning For Behavior Modeling in Video Games [19]. They 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. Under this system they have successfully been able to learn the combat behaviors of an expert player and apply them to an agent in a modified version of the video game Soldier of Fortune 2. First they extract data by observing an expert player playing the game, then by defining the feature set with common actions that they believe of great importance, such as closest goal, closest enemy, directions e.t.c they learn from this data collected. In a bit more detail they record whether or not the player accelerates, changes movement, changes facing, or jumps This part of the feature vector represents the decision made by the player. Next with the input features as well as the decision, we have a complete feature vector. This feature vector is saved and the collection of samples becomes our training and testing sets used for applying the learning algorithms. ANN with the basic back-propagation algorithm was used in this project As mentioned above we don't need the agent to be competitive for performance testing so we didn't enhance it in such a way. Also one possible problem with the mapping method for testing is that several play sessions are needed to get enough samples for the NN's to produce minimal errors. Quoting the writer; "Data was collected over the course of several game sessions and combined into one massive data set of approximately 6000 examples. Each game was run by the same expert player, whose performance was fairly consistent". This could prove quite an overkill for testing as some can argue that if several game sessions are completed the game is already tested.

A.3.12 Reinforcement learning. Reinforcement learning (RL) comprises a set of algorithms and techniques focused on maximizing an accumulated discounted reward over a period of time in

response to a series of actions performed within an environment. Through exploration and exploitation of a state-action space, based on feedback from interactions with an environment, a control policy can be learned that maximizes the reward for a sequence of actions without requiring explicit training from a domain expert [37]. RL seems really promising since the complexity of digital games coexists with the nature of exploration-based learning through an agent's interactions with its environment, underpinned by the theory of sequential decision processes, would seem complementary to the creation of game AI. We used it in a bit of a different way as described in the paper for reinforcing retest of certain regions.

A.3.13 Apprenticeship learning. "When teaching a young child to play a game, it is much easier and more practical to demonstrate it to the child rather than listing all the game rules. Then the child will learn by trying to mimic the demonstrator's performance and grasp the task gradually" [20]. Learning from an expert by watching, imitating, or from demonstration is called apprenticeship learning (AL), which employs Inverse Reinforcement learning (IRL) to solve such a problem. [36] Next we will take a look at Apprenticeship learning and IRL with the paper Learning a Super Mario Controller from Examples of Human Play [20]. They demonstrate that by using apprenticeship learning via Inverse Reinforcement Learning, it is possible to get an optimal policy which yields performance close to that of an human expert playing the game Super Mario, at least under specific conditions. To do that they use Markov Decision Processes. In machine learning, the environment is typically represented as a Markov Decision Process (MDP) and decisions made by the agent are called a policy (a probability distribution for selecting actions at each state). The goal of the agent is to find the optimal policy , a policy that maximizes accumulated rewards over time. What they do is to derive a reward function by thinking of an expert as an agent trying to maximize a hidden reward function R, which can be expressed as a linear combination of known features from the environment:

$$R^*(s) = w^* \cdot \phi(s)$$

Where R^* is a "true" reward function ,ϕ is a vector matrix of features and s represents states. Last w is a vector which specified the relative weights between these features corresponding to the reward function. The algorithm solves a linear programming problem by bringing the policy π of a RL algorithm close to $\pi\varepsilon$, the optimal policy performed by expert. Apprenticeship learning seems promising for the future. But still the issue that multiple playthroughs have to be completed in order to have sufficient data persists. Six playthroughs were done from an expert player in order to have sufficient results. Also note that Mario is a 2D game which tones down the complexity of feature extraction quite a lot.That makes it unappealing for testing agents in VR at the moment. Nevertheless a method that learns from a human tester could potentially be even better than our approach. Right now to the best of our knowledge it seems quite unstable so we didn't follow this route.

A.3.14 Testing Methodologies.
Following into our literature review, we will take a look at testing and it's automation for normal software engineering and see how the logic behind it can help us in our project. Let's start with an introduction to unit testing. In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use [10]. Unit testing is commonly automated, but may still be performed manually .The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits such as finding problems early, simplify integration for the system and facilitating changing allowing the programmers to refactor code easily. [53]

A.3.15 Random and Directed Testing. Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail . Advantages are that it quickly generate a lot of tests, ease of implementation, scales to large software applications, and reveals software errors. The problem is that it tends to generate many tests that are illegal or that exercise the same parts of the code as other tests, thus limiting its effectiveness. [23] Next we take a look at Directed random testing [35]. Directed random testing is a new approach to test generation that overcomes these limitations, by combining a bottom-up generation of tests with runtime guidance. A directed random test generator takes a collection of operations under test and generates new tests incrementally, by randomly selecting operations to apply and finding arguments from among previously-constructed tests. As soon as it generates a new test, the generator executes it, and the result determines whether the test is redundant, illegal, error-revealing, or useful for generating more tests. The technique outputs failing tests pointing to potential errors that should be corrected, and passing tests that can be used for regression testing. In a similar way we have our agent wander with a randomized pattern in the environment executing test cases according to stimulus or direct him towards operations and arguments .

A.3.16 Combinatorial Testing. As the complexity of system grows ,testing requirements became immense. Testing all the inputs for a program and their combinations is an outstanding task and that's when Combinatorial Testing rises. Combinatorial Testing (CT) can detect failures triggered by interactions of parameters in the Software Under Test (SUT) with a covering array test suite generated by some sampling mechanisms. CT has the following characteristics: CT creates test cases by selecting values for parameters and by combining these values to form a covering array. The covering array specifies test data where each row of the array can be regarded as a set of parameter values for a specific test [2]. Here we will give an example of combinatorial testing in games, found in the above mentioned paper. Suppose we want to test a network game software running in the Internet environment. The operation of this game may be influenced by many parameters, such as browser, operating system, the type of network access, graphics, audio, the number of players, and so on. Each of these parameters may take on many possible values. The interactions of these parameters may cause some failures. Due to the large combination space, exhaustive testing by testing all the parameter value combinations is generally

impractical. Even if we have the resources to try all value combinations, this is not effective because most of the value combinations do not cause any failure. CT provides a practical way to detect failures caused by parameter interactions with a good trade off between cost and efficiency. It samples the large combination space using a smaller test suite to cover certain key parameter value combinations. Combinatorial testing will probably not be integrated in our testing scenarios , but in a larger scalability it would become really useful and interesting as future work.

A.3.17 Automation in testing for FIFA. Last stop in our literature for testing is the paper; Software Testing by Active Learning for Commercial Games [18] This work focused on creating an active learning framework for blackbox software testing. The active learning approach samples input/output pairs from a blackbox and learns a model of the system's behavior. This model is then used to select new inputs for sampling. They used the popular game FIFA of EA for their research and focused on corner kick testing. Lets dive a bit more into detail about the framework which is based on "semi-automated gameplay analysis" (SAGA). The game engine is treated as a black box and SAGA-ML interacts with it through an abstraction layer. This layer is game-specific and translates game specific data and function calls to an abstract state format. The sampler component uses the abstraction layer to evaluate situations by running the game with an initial state and a sequence of actions, and then observing the outcome. The learner uses the data gathered by the sampler to construct a concise model (or summary) of the game's behavior. The learner may then request more samples to refine its model. Together the sampler and learner form the active learning part of the system. Finally, the learned model is passed to the game-specific visualizer for the designer to evaluate. They use a rule based system for the learning part. For example, in the shooter-goalie scenario, such a rule might be: IF the shooter is within 5 meters of the goalie AND the angle between shooter and goalie is between 30 and 40 degrees AND the goalie is within 1 meter of the goal's center THEN the probability of scoring is greater than 70 % To sample all the regions and decide what to sample next they used and evaluated different active learning algorithms and even developed one of their own. In their own method called Decision Boundary Renement Sampling the rules describe rectangles where a prediction like the above described is positive (true) or negative (false) (this is just an example, predictions could be more complex, e.g. probabilities). If there are several rules that overlap, but all agree in their prediction, then they can merge them together to form a region. To evaluate that a region's boundary is correct, new samples are randomly placed on both sides of the boundary within a small margin. Using automated black box methods to learn specific parts of the game and creating an active learning framework for automated software testing seems to work for commercial games and help in various design elements and balancing of the overall game. Since we won't use a commercial game and our focus is performance testing and not testing that helps game designers, a complete framework is not needed. As such it was not created but a thought of a framework for testers with no coding experience in a similar way could be interesting future work.

A.4 Defining Game Components

There is no extensive literature of identifying specific game components for different game genres. Game development is usually an ad-hoc engineering process which is characterized by low level development due to the unique nature of different games. One such effort and notable mention is the paper Improving Digital Game Development with Software Product Lines [1]. They integrate the idea of software product lines into game development and try to create domain specific language (DSL) and core architectures for 2D arcade games. They also provide advices for analyzing game domains, with some important mentions being; Select domain samples ,defining and refining game domain features ,create sub domains and anticipating future features. They give special focus on the subdomain part mentioning that they don't believe in a one-size-fits-all game architecture. They advise to consider partitioning the target game domain into subdomains (for example, partitioning a broader arcade domain into shooter and maze subdomains). The individual analyses of more specific subdomains lead to more expressive and effective SPL assets, such as DSLs. We also tried to create functions that the agent will follow like "Shoot", "Collect" e.t.c that could potentially work in a DSL format and be universal across FPS or VR games. They could even be defined as part of "agnostic behavior" that is be common in most recent games (e.g look around) while everything else as "game specific" logic which describes gameplay elements (e.g. throw grenades). This paper complements the DSL related work of the paper and together make a nice consideration for future work.

A.5 Experiment Details

The experiment was conducted via the online freelancer platform Upwork [54]. We listed the experiment as a paid job to attract freelancers with testing skills.

A.5.1 Job Posting.

Needs to hire 5 Freelancers

Looking for testers to play an experimental Virtual Reality game and report any performance issues found. Task is fairly simple so no extreme experience is required

Tester must have:

- A decent Android 6.0+ phone
- A virtual reality mask for mobiles (e.g google cardboard)
- A wireless mobile controller

The whole procedure should take less than an hour and pays between 10-15$

- One-time Project: Find a bug
- Project Type: One-time project

You will be asked to answer the following questions when submitting a proposal:

- What android phone do you have? Do you have a wireless controller for mobile?

A.5.2 Consent Form.

A consent form was used to inform the participant about experiment to warn him/her about the risks of VR.

Consent form

Experiment: VR performance testing
Experimenter: Georgios Kokkinos
Affiliation: Utrecht University

You are invited to participate in a research study that tests a VR FPS setup. In the experiment, you will be wearing a headmounted display. You will explore a VR world and report any performance issues found.

Risks: The headmounted display may cause temporary nausea and temporary dizziness in some users. The procedure should be stopped immediately when you indicate you are experiencing these or any other type of discomfort.

Your individual privacy will be maintained in all published and written data resulting from the Study.

If you agree with the above stated conditions and are willing to participate in the experiment, please sign below. By signing the form, you confirm that you meet the following conditions:

- You have read the above consent form, understood it and you agree to it.
- You want to participate in the above mentioned Experiment.

A.5.3 Instructions.

Specific Instructions were given about the testing procedure:

(1) Connect your phone with the PC and run the .bat file. The game will be installed. (you need usb debugging on, transfer files mode and trust any sources on settings)
(2) If it doesn't you will have to install the apk on the of the zip folder manually.
(3) Check the "rooms.pdf" file so you can have a general idea about the environment.
(4) On the entry level move over the letters "manual play" and the game will start.
(5) Time your playthrough and stop when you think you found all the performance drops.
(6) Play through the game with the goal to find any performance drops.
(7) When you find a performance drop please note down the room that you found it or/and the action you did to cause it.
(8) Complete the questionnaire on: https://goo.gl/forms/AlnCVkHbuBCGb8uB2

Game instructions for gameplay help

The game in general plays as a first-person shooter. The main objective is to collect the statues (figure 5), beat the enemies and reach the final goal (figure 6). You will need to rotate a lot so find a comfortable space to play.

Fig 5. Statue / Collectible in the game

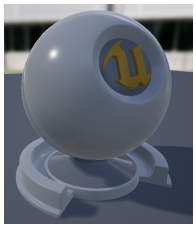

Fig 6. End goal

Fig 7. Control Representation

(1) Use the left thumb stick or dpad to move around
(2) Rotate your head to look around
(3) Press the X* button to collect statues or open Doors
(4) Press the A* button to shoot (You can also shoot with the trigger button if your controller has one it).

**According to figure 7.*

A.5.4 Room Overview

. All the rooms were provided in pictures for the testers to check and be able to report errors based on them. For the retesting experiment only the pictures with the performance drops where shown.

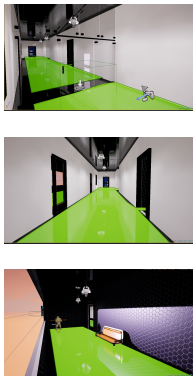

Fig 8. Example of pictures of rooms provided to the testers (room1, room2, room3).

A.5.5 Questionnaire.

After the testers finished the game the participants answered some general questions and reported any issues found via a questionnaire.

VR Performance Testing Questionnaire

This questionnaire was made to help report any performance drops found on the experimental VR game ATA_UU and collect data about the testers.

Name *

Short-answer text

Age *

Short-answer text

Gender *

- Female
- Male
- Prefer not to say

Do you have experience in testing? *

- Yes
- No

If yes, please specify your experience level.

- up to 1 year
- 1-3 years
- 3+ years

Game data

Play through the game and answer the following questions. Please don't forget to time yourself

Please specify in which rooms you found performance drops (pictures of rooms can be found on the pdf file that I sent you).

	Drop found
Room 1	
Room 2	
Room 3	
Room 4	
Room 5	
Room 6	
Room 7	
Room 8	
Room 9	
Room 10	
Room 11	
Room 12	
Room 13	
Room 14	
Room 15	
Room 16	

Add any comments about the previous question, if any.

Long-answer text

Did you find any performance drops while interacting with the environment? *

- Yes
- No

If yes please specify (e.g I noticed a performance drop while bumping into a chair on room 3).

Long-answer text

Did you find any other performance drops?

Long-answer text

How much time did it took you to find all the performance drops? *

Short-answer text

Upload the file with the frame-rate information found on the folder of the game AFTER you completed the game once. Name of folder is: "Internalstorage\UE4Game\ATA_UU\ATA_UU\Saved\Profiling\FPSChartStats\..."

ADD FILE

As we see on the last question of the questionnaire the users were asked to send a csv file that is generated when they complete the game. This file contains information about the performance. We use it to create graphs for ease of comparison between the testers, either human testers or agents-AI testing bots. The graphs are created by a windows form application [55] we developed. By passing the csv generated file, a graph is created, showing the rendering thread (RT) and game thread (GT) computational time required for the frame to be processed. Our graph creator works without problems for different games. An example of it's graphical interface can be seen on figure 9.

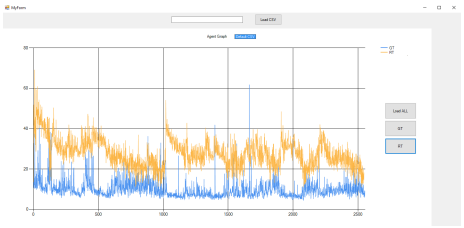

Fig 9. You can load a CSV by pressing the button LoadCSV while adding it's name on the text box next to it. Using the buttons GT and RT the graph lines appear accordingly. On Y axis we can see the time required for processing and the X axis shows the number of the frame.

A.6 Research Approach additional information

In this section we provide more information about our implementation through some blueprints examples that also show how to reproduce the algorithmic steps of the retesting approach discussed on the paper.

One of the major step is learning the rewards and creating the reward matrix. This serves as the first step in our implementation of retesting and is controlled with a boolean value called "learnRewards" (on the Level blueprint/script).

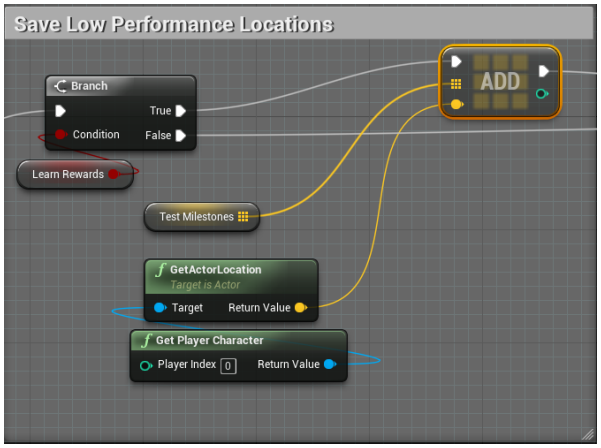

Fig 10. Blueprint Example

As seen in figure 10, if the learnRewards is set to true we save the locations to the array "TestMilestones" that the performance drop happened by using the global location of the player at that point. So for the first step of the retesting experiment this boolean should be set to True.

After the reward matrix is created we use an external q learning algorithm to produce the Q matrix. Our works is heavily based on John McCullock's implementation of path planning using q learning[28]. It's a python implementation that follows the principles described on the paper. On figure 11 we see an example of code that updates the $Q(i,j)_{t+1}$ using the equation described in the paper.

```
11  def update_q(state, next_state, action, alpha, gamma):
12      rsa = r[state, action]
13      qsa = q[state, action]
14      new_q = qsa + alpha * (rsa + gamma * max(q[next_state, :]) - qsa)
15      q[state, action] = new_q
16      # renormalize row to be between 0 and 1
17      rn = q[state][q[state] > 0] / np.sum(q[state][q[state] > 0])
18      q[state][q[state] > 0] = rn
19      return r[state, action]
```

Fig 11. Code example of Q learning implementation

The results of the Q matrix (figure 11) as discussed on the algorithmic steps of the paper are then passed via a data table to the agent and we can run the retesting experiment.

```
[[  0.00000000e+00   1.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00]
 [  1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   1.00000000e+00   0.00000000e+00   0.00000000e+00
    1.01179282e-11   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   7.99997747e-01   0.00000000e+00
    2.00002298e-01   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   5.23043811e-01   3.51027399e-01
    0.00000000e+00   1.25928774e-01]
 [  0.00000000e+00   0.00000000e+00   7.70691931e-01   1.74821585e-01
    5.44864200e-02   0.00000000e+00]]
```

Fig 12. Converged Q matrix

By setting to true the boolean named "followQMatrix" (on the AIController blueprint/script) the agent prioritizes the milestones accordingly (if we don't want to prioritize milestones it should be set to false, which is by default). To do that we multiply the default desire value of the milestone with the times a milestone is visited in the previous step.

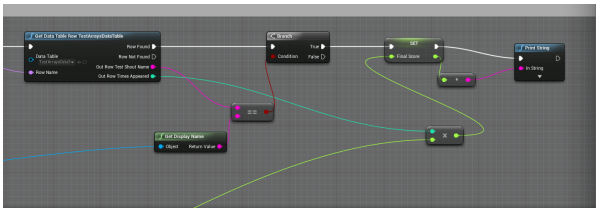

Fig 13. Blueprint Example 2

We can see in figure 13 that the results of the data table are used using the game engine built node "Get Data Table Row TestArraysDataTable". We check the name of the milestone perceived and we multiply its score with the integer "timesAppeared". Then we set the variable "finalScore" and its added on the perceived object list. The full code is commented either it's blueprint based , c++ or python scripts. It's online and available for review.

Note that the game and the agents are implemented with version of engine 4.16.3 but it should run successfully with newer versions of the engine with no to minor adjustments. Last if an android build is needed the system should also have the NVPack installed which is a tool that contains multiple sdk's which are needed for building android applications [31]. The whole project though can also be used on the pc or the editor of the game engine without building it externally.

REFERENCES

[1] W.B. Furtado Andre, L.M. Santos, and L. Ramalho Geber. 2011. Improving Digital Game Development with Software Product Lines. (2011).
[2] Nie Changhai and Leung Hareton. 2011. A survey of combinatorial testing. (2011). http://dspace.mit.edu/handle/1721.1/53297
[3] Woei-Kae Chen. 2016. A Game Framework Supporting Automatic Functional Testing for Games. (2016).
[4] Lim Chong-U, Baumgarten Robin, and Colton Simon. 2010. Evolving Behaviour Trees for the Commercial Game DEFCON. (2010). https://pdfs.semanticscholar.org/2763/7d4de11b3d6baeb9bf26597ec6fd86a51d5a.pdf
[5] Simpson Chris. 2014. Behavior trees for AI: How they work, Gamasutra article. (2014). https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php
[6] Redavid Claudio and Farid Adil. 2011. An Overview of Game Testing Techniques. (2011). http://www.idt.mdh.se/kurser/ct3340/ht11/MINICONFERENCE/FinalPapers/ircse11_submission_15.pdf
[7] Andrew G. Barto d S. Sutton. 1998. Reinforcement learning: an introduction. The MIT Press. (1998).
[8] Frank Dabek. 2002. Event-driven Programming for Robust Software. (2002).
[9] Unreal Engine documentation. [n. d.]. How Unreal Engine 4 Behavior Trees Differ. ([n. d.]). https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html
[10] Huizinga Dorota and Kolawa Adam. 2007. Automated Defect Prevention: Best Practices in Software Management. (2007).
[11] A. Elyasov, I.S.W.B. Prasetya, and J. Hage. 2013. Guided Algebraic Specification Mining for Failure Simplification. http://dx.doi.org/10.1007/978-3-642-41707-8_15. (2013).
[12] A. Elyasov, I.S.W.B. Prasetya, J. Hage, and A. Nikas. 2014. Reduce First, Debug Later. http://dx.doi.org/10.1145/2593501.2593510. (2014).
[13] Unreal Engine. [n. d.]. Introduction to C++ Programming in C++. https://docs.unrealengine.com/latest/INT/Programming/Introduction/. ([n. d.]).
[14] Unreal Engine. 2017. Documentation BT Unreal. (2017).
[15] Unreal Engine. 2017. Documentation garbage collector. (2017).
[16] ESA and Entertainment software association. 2017. Essential facts about the computer and video game industry. http://essentialfacts.theesa.com/mobile/. (2017).
[17] VR Games for. 2017. List of virtual reality games. https://vrgamesfor.com/list/. (2017).
[18] Xiao Gang and Wilkinson Dana. 2005. Software Testing by Active Learning for Commercial Games. (2005). https://www.aaai.org/Papers/AAAI/2005/AAAI05-142.pdf
[19] Ben Geisler. 2017. Integrated Machine Learning For Behavior Modeling in Video Games. (2017). https://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-012.pdf

[20] Lee Geoffrey, Luo Min, and Zambetta Fabio. 2014. Learning a Super Mario Controller from Examples of Human Play. (2014). https://titan.csit.rmit.edu.au/~e46507/publications/min-mario-cec14.pdf

[21] David E. Goldberg. 1998. Genetic Algorithms In Search, Optimization, and Machine Learning. (1998).

[22] IEEE. 1990. IEEE Std 610.12-1990,pp. 1–84,. (1990).

[23] J. Marciniak John. [n. d.]. Random Testing chapter in Encyclopedia of Software Engineering. ([n. d.]).

[24] Matt Klingensmitt. 2013. Overview of Motion Planning. (2013).

[25] J. Hettinger Lawrence, S. Berbaum Kevin, and S. Kennedy.. Robert. 1990. Vection and Simulator Sickness. (1990).

[26] Galway Leo, Keith Charles Darryl, and M. Black Michaela. 2008. Machine learning in digital games: A survey. (2008).

[27] C Lewis. 2010. What Went Wrong: A Taxonomy of Video Game Bugs. (2010).

[28] John McCullock. [n. d.]. Q learning tutorial. http://mnemstudio.org/path-finding-q-learning.htm. ([n. d.]).

[29] Tom Mitchell. 1997. Artificial Neural Networks, Chapter 4. (1997).

[30] Jaklin Norman, Cook Atlas, and Geraerts Roland. 2013. Real-time path planning in heterogeneous environments. (2013). http://www.cs.uu.nl/docs/vakken/mpap/papers/7.pdf

[31] Nvidia. [n. d.]. NVIDIA CodeWorks for Android. https://developer.nvidia.com/codeworks-android. ([n. d.]).

[32] Stanford Logic Group of Stanford University. [n. d.]. General Game Playing. http://www.general-game-playing.de/. ([n. d.]).

[33] Souissi Omar, Duvivier David, and Artiba Abdelhakim. 2013. Path planning: A 2013 survey. (2013).

[34] Jeff Orkin. 2004. Agent Architecture Considerations for Real-Time Planning in Games. (2004).

[35] Carlos Pacheco. 2009. Directed random testing. (2009). http://dspace.mit.edu/handle/1721.1/53297

[36] Abbeel Pieter and Y. Ng Andrew. 2004. Apprenticeship Learning via Inverse Reinforcement Learning. (2004). http://ai.stanford.edu/~ang/papers/icml04-apprentice.pdf

[37] S. Sutton Richard and G. Barto Andrew. 2004. Reinforcement Learning: An Introduction. (2004). http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf

[38] U. Rueda, T.E.J. Vos, and I.S.W.B. Prasetya. 2015. Unit Testing Tool Competition ÂŰ Round Three. http://dx.doi.org/10.1109/SBST.2015.12. (2015).

[39] Wikipedia s. 2017. Virtual reality headset. https://en.wikipedia.org/wiki/Virtual_reality_headset. (2017).

[40] Karakovskiy Sergey and Togelius Julian. 2011. The Mario AI Benchmark and Competitions. (2011). http://julian.togelius.com/Karakovskiy2012The.pdf

[41] Helge Spieker. 2015. REINFORCEMENT LEARNING FOR SOLVING SHORTEST-PATH AND DYNAMIC SCHEDULING PROBLEMS. (2015).

[42] Helge Spieker. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. (2017).

[43] Wouter van Toll, F. Cook Atlas, and Geraerts Roland. 2011. Navigation meshes for realistic multi-layered environments. (2011).

[44] T.E.J. Vos, P. Tonella, J. Wegener, M. Harman, I.S.W.B. Prasetya, E. Puoskari, and Y. Nir-Buchbinder. 2011. Future Internet Testing with FITTEST. http://www.academia.edu/download/43917809/csmr2011.pdf. (2011).

[45] Occulus VR. 2017. Oculus Best Practices. (2017). https://static.oculus.com/documentation/pdfs/intro-vr/latest/bp.pdf

[46] VRHeads. 2016. Motion in VR. (2016).

[47] Brooklyn Waters. 2015. Physics and Frame Rate: Beating motion sickness in VR. (2015). http://mtechgames.com/downloads/PhysicsandFramerateBeatingmotionsicknessinVR.pdf

[48] Wikipedia. 2017. Behavior trees, Wikipedia. (2017).

[49] Wikipedia. 2017. Cardboard VR, Wikipedia. (2017).

[50] Wikipedia. 2017. Dijkstra's Algorithm. https://vrgamesfor.com/list/. (2017).

[51] Wikipedia. 2017. Navigation Mesh, Wikipedia. (2017).

[52] Wikipedia. 2017. Selenium software. https://en.wikipedia.org/wiki/Selenium_(software). (2017).

[53] Wikipedia. 2017. Unit Testing, Wikipedia. (2017).

[54] Wikipedia. 2017. Upwork Freelancer Platform. https://en.wikipedia.org/wiki/Upwork. (2017).

[55] Wikipedia. 2017. Windows Forms Application. https://en.wikipedia.org/wiki/Windows_Forms. (2017).

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