Sampo Tuisku

Automated Regression Testing for Cloud Based Mobile Games

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Information and Communications Technology

Game Applications

Bachelor’s Thesis

31 October 2020

Author Title Number of Pages Date	Sampo Tuisku Automated Regression Testing for Cloud Based Mobile Games 40 pages 31 October 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructor	Antti Laiho, Senior Lecturer

Automated testing is crucial for cloud-based gaming services due to the large number of games that they provide. On each new release of the service, the games must be tested for compatibility with the new release. Due to the complexity of games, issues might arise in the games that are on the service. With an increasing number of games on the service, manually testing becomes cumbersome and costly. The objective of this project was to find and create a way to test these games rigorously, in-detail and autonomously. The project was done for a mobile cloud gaming company, Hatch Entertainment Ltd.

To try to solve problems concerning compatibility and testing, a script was created to manually record all game's lifecycle's touch events and frames. Using these touch events, the new releases of the service's games could be played autonomously, and its frames could be recorded for comparison. With these autonomously captured frames we could compare them to the manually recorded ones and get the difference for possible malfunctions.

Prototyping was done using Python and later on it was implemented into the existing cloud service SDK. During the scope of this project, this solution was proven to be probable but not yet implementable due to the complex nature of the given issue. The main issues derived from a lack of accuracy when comparing the automated and manual session with each other and performance problems when repeating user inputs. With the prototype implementation and ideas for solving the issues along with it, the project is likely to benefit the case company by increasing the efficiency and accuracy of the automation process.

Keywords	games, cloud gaming, cloud service, automation, image recognition, image comparison, mobile, regression

Tekijä Otsikko Sivumäärä Aika	Sampo Tuisku Automatisoitu regressiotestaus pilvipohjaisiin mobiilipeleihin 40 sivua 31.10.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Antti Laiho

Automaatiotestaus on erityisen tärkeää videopelien pilvipalveluille pelien ison määrän vuoksi. Kaikkien näiden pelien yhteensopivuus pilvipalvelun kanssa tulee testata uudelleen aina palvelun uuden version julkaisun jälkeen. Uusien julkaisujen myötä näissä peleissä saattaa ilmetä ongelmia. Kun pelit pilvialustalla lisääntyvät, manuaalisen testauksen tarve kasvaa ja testausprosessi hidastuu. Insinöörityön tavoitteena oli löytää ratkaisu nopeuttaa tämä testiprosessi automatisoinnilla. Työ tehtiin suurelle kansainväliselle yritykselle.

Testausongelman ratkaisemiseksi kirjoitettiin skripti, joka nauhoittaa toimivan version pelin manuaalisen pelisession kosketukset ja kuvat. Käyttämällä näitä tallennettuja kosketuksia aina uuden version julkaisun yhteydessä pelit pystyttäisiin pelaamaan automaation avulla ja samalla tallentamaan myös kuvat. Vertaamalla automaation kuvia manuaalisen pelisession kuviin uuden version julkaisun mahdollisesti aiheuttamat ongelmat voitaisiin löytää helposti ja nopeasti.

Testiautomaation prototyyppi kehitettiin Python-ohjelmointikielellä, ja myöhemmin se yhdistettiin pilvipalvelun koodiin. Työtä tehdessä todettiin, että tämä ratkaisu voi olla mahdollinen, mutta ei vielä testituloksien kannalta järkevä ongelman kompleksisuuden vuoksi. Projektin pääongelmat ilmenivät automatisoidun ja manuaalisen session testaustarkkuudessa ja kosketustapahtumien toiston tehokkuudessa. Testiautomaation implementaation ja ideoiden avulla asiakasyritys voi hyötyä tuloksista tulevaisuudessa, jos yritys haluaa tätä testiautomaatiotyötä jatkaa.

Avainsanat	pilvipalvelu, regressio, automaatio, mobiilipelit

Contents

List of Abbreviations

ADB	Android Debug Bridge. Command-line tool used for interacting with an Android device.
PvP	Player versus player. Player interacting and conflicting with another player.
USB-cable	Universal serial bus cable. Used for connecting devices.
WiFi	Wireless Fidelity. Provides wireless internet connection.
QA	Quality Assurance. Quality assurance is a representation of making sure the thing being worked on reaches the quality standards set.
APK	Android application package. Android operating system file format.
OBB	Opaque binary blob. File format used for storage of large files.
CNN	Convolutional neural network. A class of deep neural networks.
ML	Machine learning. Algorithm that improves itself.
AI	Artificial intelligence. Machines intelligence.

1 Introduction

This thesis is performed in collaboration with Hatch Entertainment Ltd (Hatch). The goal of this thesis is to investigate the ability of a prototype to automate regression tests more accurately and more thoroughly for mobile video games on the Hatch Cloud Service App using automated playing and image recognition. The evaluation of the prototype is done with respect to its expected future cost and time savings and on its ability to test the games to a further extent and higher accuracy than the previous automated testing method.

Hatch is a cloud gaming service (“the service”) which provides over a hundred online games for players all around the world. Manually testing these games’ compatibility with the service is slow and expensive, especially when there are multiple operating systems that the service is provided on. Manual testers are still an important resource to the game development industry, since they are still the most reliable testing method there is. The biggest issue with manual testing is that it is timely, costly and inefficient. The solution to improve the overall testing performance is to technically automate it. Unfortunately, though, automated testing is not yet advanced enough that it could completely replace manual testing. This is especially relevant in video games development processes. In the case of Hatch, every time there is a new release of the application all existing games on the service must be tested and their compatibility with the new release needs to be reevaluated. This is a common occurrence during the lifetime of the service and therefore automated tests are required in order to speed up the development process.

In cloud-based services like Netflix and Spotify, testing the data on each new release is straightforward due to the type of data being tested. Audio and video contents have formats that are standardized, whereas each video game can differ in engineering from one another. In comparison to cloud-based video and audio platforms, gaming streaming services contain data that differs from case to case. Each game is inherently built differently and is programmed using different tools, APIs and programming languages; therefore, each game reacts differently to each system.

Each game is manually tested carefully the first time they arrive on the service. Once the game has proven to work internally it can be shared for the players using the application. Regression smoke testing is a great way to check that the games are starting as normal and taking input and responding to it after creating a new release of the application. In most cases this is enough to prove that the game works but due to games not being constant streams of one action like audio and video, they can break not just at the start but also later during their lifecycle on the cloud platform. This is why regression smoke testing might not be enough to spot out new issues arriving along the new release. Manually doing this the tester often can find possible issues within a few minutes of gameplay. This does not prove that there would not be new bugs occurring later in the game or even at the end of the game hours into the gameplay. If the tester played all the games for an hour, testing would take weeks and months. In an optimal situation the game would be played from start to finish to secure the flawless functionality of the game. If rigorous testing is not done, then the customers playing the games might discover bugs and abandon the usage of the service altogether and even leave bad reviews about the functionality of the app. In the end this can lead to customer churn.

Automated testing provides more accuracy of the software development process and it gives the possibility to replace long manual test runs for the games on the service. The automation offers in-depth results and data of where and why the game broke on the service. This data is then handled and manually checked by the quality assurance team and the issue is analyzed and fixed accordingly.

In an ideal world the implementation of the presented prototype automation could provide a possibility for more accurate testing results than what a human could ever achieve, it could speed up the process of testing by days, it could cut down costs and it could also open up more time for existing testers to test new and up-and-coming games for the service and eliminate the chaos of manually testing hundreds of games on each release of the service.

This thesis investigates the latest trends of automated testing techniques and it discusses in detail one such prototype I programmed for Hatch. This thesis also explores the possibility of using newly emerging technologies for automated testing purposes, such as audio testing and image comparison using machine learning.

The outline of the thesis is as follows. The second section does an overview of the business context the thesis was written in. The third section outlines the essential theory behind automated testing practices, latest trends and techniques. The fourth chapter goes into the details of the features of the example prototype script, as well as its description of the Python implementation and finally its results. Sixth chapter is for discussing possible improvements. The thesis is then concluded in section seven.

2 Business Context and Automation Testing in Cloud Mobile Games

Hatch [1] is a subsidiary of Rovio Entertainment, established in late 2016 in Espoo, Finland. Hatch is a mobile cloud gaming service that provides premium mobile games on demand all in one application. In addition to games being played on the cloud, Hatch provides social features such as leaderboards, competitions, real-time PvP matches as well as a version for kids to play in an educative and safe environment without ads and in-app purchases. Figure 1 shows and example of the kids application.

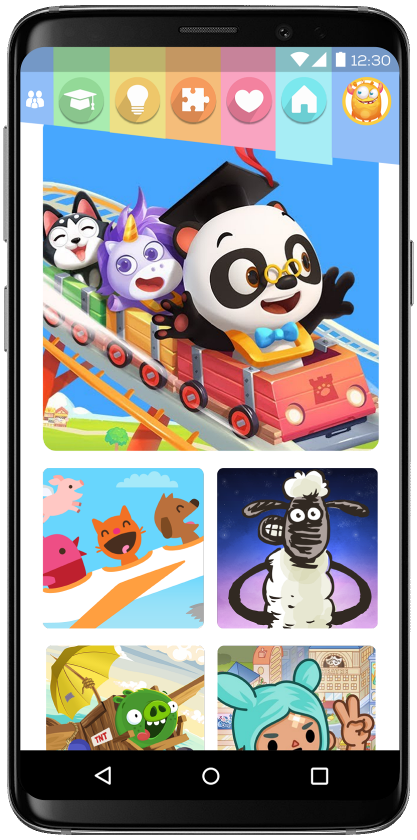

Figure 1. Hatch kids application front page. [27]

Hatch is a pioneer in cloud-based gaming services and has attracted countless large game companies around the globe. Some of these “games on Hatch include Arkanoid Rising, Monument Valley, Mini Metro, Party Hard Go, Beach Buggy Racing, Evoland, Crashlands, Angry Birds GO! Turbo Edition, and Shaun the Sheep - Home Sheep Home - Farmageddon Party Edition. All are available to play instantly without downloads and without in-game purchases”. [26]

2.1 Benefits of Streaming Games

One of the biggest benefits of streaming games is the fact that one does not need to download anything but the main application itself. Hence, instead of downloading any number of games on the device independently, one has access to unlimited games on the cloud. This frees up space on the device to use for different purposes. If one does not like the game they are playing, they do not have to delete the game and access the device store to download a new one, but instead all that must be done is press play within the cloud service application. This is all possible within Hatch’s service.

The gaming industry is growing larger every year and the number of games is increasing constantly, making it harder to find the best games of one’s liking. This is an opportunity for streaming services to provide a handpicked selection of premium games for their platform, making it easier for the common player to find the best options on the market with just one click away.

2.2 Business Challenge, Objective and Outcome

Nowadays the quality standards of gaming are high and people who play games are aware of it. If these standards are not met, the feedback is instantly bad and that affects the status of the company and the people who play its games. Therefore, as a frontrunner in the cloud gaming industry, quality and maintaining player experience is key for Hatch. With big companies like Microsoft and Google stepping into the cloud gaming industry and having access to the most industry resources, start-ups like Hatch need to be prepared to deliver a high-quality product in an efficient manner. In order to do that, premium games, exceptional talent and fast testing protocols are needed.

Before acquiring over a hundred games on the service, regular manual testing and automated UI smoke tests were enough to spot out insufficiencies on new releases of the service. Due to the ever-growing interest in the service from the gaming industry the number of games kept increasing and therefore the demand for testing grew as well. ***The objective of this project is to explore a possible solution to solve the issue of testing hundreds of games on the service by automating game testing with a more accurate and rigorous implementation of the automation process.*** The outcome of the project is a script that records the working version of the game on the service when manually playing the game. The script will then capture real-time frames and user touch events. Once a new version of the service is provided, the game is played autonomously using the stored touch events of the working version while capturing the new frames. These two sets of frames are then compared, to find any possible issues that have occurred due to the new release of the service.

2.3 Definition of Automated Testing

Automated software testing implicates the usage of software tools to test a piece of software. However, testing automation does not exclude manual one and how much manuality is required depends on the type of software tested. In general, automation testing is necessary to modern agile software development due to the frequent new releases of a piece of software. This is especially valid for the mobile cloud gaming industry, since there are continuous changes, additions and improvements done to cloud service. With these changes and additions, regression might occur not just on the service itself but on the games running in it. A good definition of automated testing would be the usage of automation code to execute automated programmed tests, to compare the results and to record the spotted regressions.

In the commonly used software engineering methodology, agile development, automation testing is inevitably essential. The so-called ‘continuous integration’ represents automation and one way to do that is regression testing. To design a good software testing architecture, one must be very analytical and critical to the given software to be tested. Each automated testing technique serves a different purpose based on the stakeholder’s needs and the expected functionality of the code. Agile development demands good software automation tests in order to guarantee a successful deployment of the software. The below depicted diagram showcases the idea behind automated testing.

Automation tests are the collection of software tools used to technically test different computational programs. Such examples of tools are:

- Regression testing
- Smoke testing
- Analysis of log files
- Reporting of found errors/bugs and further possible improvements
- Utilities (setup/clean up test environment)
- Test generation and management of the execution files
- Test execution with defined functionality pass/fail result, unit tests

This thesis focuses primarily on regression testing, since it is an incredibly efficient tool in the context of mobile cloud gaming. The goal of automated testing in the gaming industry is essentially saving time and costs and simplifying the execution of tests. Furthermore, test automation should be a reliable and error proof source of testing software. In streaming mobile platforms that regularly have several releases in short periods of time, both the service as well as the games must work bug-free, fast, without lags. In order to achieve this, the developers need a reliable testing resource that can identify and report possible issues. Regression testing works perfectly in this case, as it is rapidly and regularly checking for any new errors in games that might have occurred since the last deployment of the software. Regression can occur deep in the games which is important to detect. The following pyramid depicts the levels of automated testing in practical use.

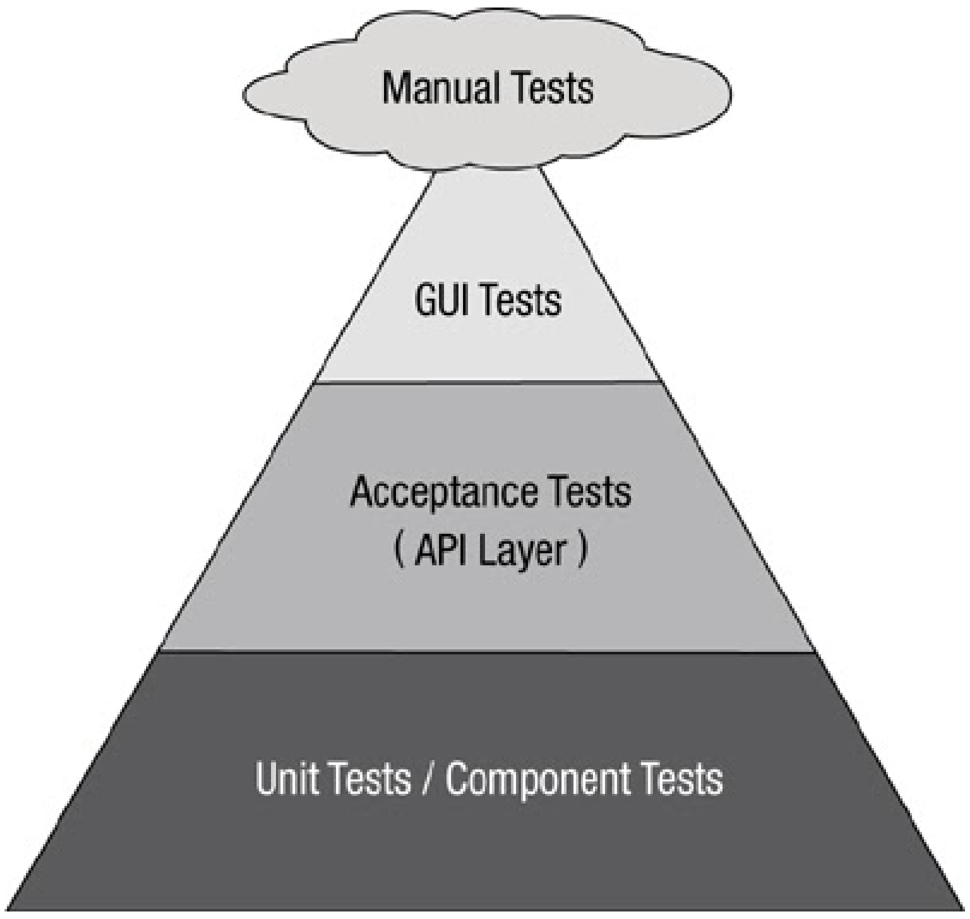

Figure 2. Pyramid of the test automation [2].

Figure 2 represents the hierarchy of the test automation layers. First and foremost, there are the manual tests which are the most reliably yet time and money consuming. Next there are the graphical user interface tests, which test the functionality on the surface level. Acceptance test are to determine needs and processes by the users and customers. Unit tests test the functionality of the methods used in the software.

2.4 Prototype Example of Automation Testing

The prototype of the new automation method aims to solve the issue of identifying regressions occurring later in the games' life cycle upon new releases of the service. The idea for this is to create a Python script that enables a computer to play the games autonomously while discovering possible regressions in the games. Later on, if the prototype shows itself to be successful, it would be implemented to the existing codebase that the service functions upon.

In order to create a script that plays the games autonomously, we need to tell the computer how to do that. This is done by recording a manual run of the game on the application that the computer can later replicate. The manual session is done on a fully working version of the game on the application. While replicating the manual run, the autonomous session will be recorded as well for comparison to the manual session. When recording the session of the game for the automation, the game has already gone through the manual testing phases and has been approved to work on the service. Therefore, every time the automation is run, we know it is comparing the results to a working version of the game and if changes arise, regression has occurred along with possible bugs and issues.

To record a working run of the game for the automation, a manual tester first plays the game along with the script that records all the touch events the tester is inputting while playing the game. The script saves these touch events onto a log file. The touch event data contains the coordinates where the tester tapped on the screen, if the tester was holding, swiping or touching the screen and it also stores the timestamp when the action occurred during the recording. These touch events can be later repeated to play the game autonomously.

For spotting any regression, we need to feed the computer enough data to which it can compare the result to the working run of the game. This data is recorded along with the touch events. We record every frame of the playthrough as an image. This is simply done by recording a video of the playthrough and later on the video is divided and cropped into several frames. This opens up the possibility to compare results using the video or the cropped frames. For the purpose of this specific implementation we are using the frames for comparison.

Once the tester has played a session of a game with the script recording the frames and the touch events, the data is saved in a database for future test runs of the automation. Using this data, the script should be able to play the games autonomously. At the same time, the script is recording new data for comparison against the already existing data that was recorded during the prior optimal run. Hence, we obtain two sets of frames that are then used for detection of possible errors in the games’ software.

Every time there is a new release of the service, automation testing ensures that the new code is error-proof and functions as expected. In order to do so, there is a need to detect any possible regression in the games uploaded on the newly updated platform. The tests perform as follows: the two data sets of frames that are returned by the script, are compared to each other and the tests analyze the results of it. The automated tests compare the two sets of frames and check whether two same positioned frames from both sets differ one another. If such a difference is identified, we can say that a regression was just detected, and such an event is notified and processed accordingly. If a regression occurs, the two identified different frames are then carefully compared to each other. Figure 3 gives a visual representation of the comparison process. In this figure, frame 102 has a graphical error in the automated session which means regression has occurred.

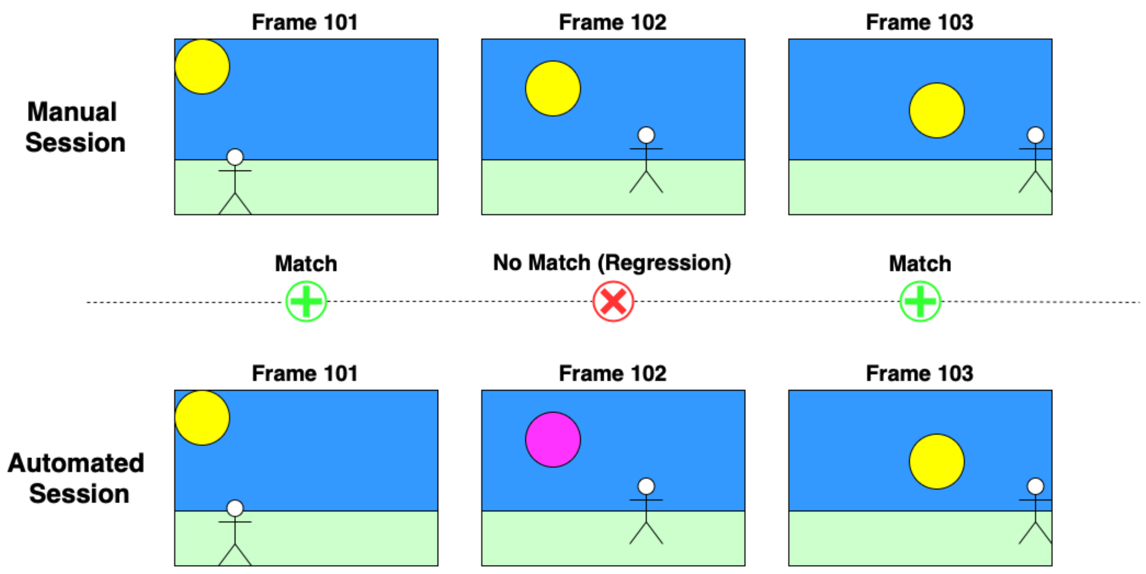

Figure 3. Visualization of comparing frame sets. Frame 102 of the automated session has a graphical error which was detected as regression.

The last procedure done by the automation is a test report that includes all necessary information about the detected regression and how the frames are different from one another. In this test report all the non-matching frames are visually presented for the manual tester to evaluate. In the end it is the tester's duty is to certify whether there actually is a regression or not.

In an ideal case for easily understandable test reports, the script will tell exactly what the difference between the two frames is and what type regression has occurred. The automation will run every game on the service, compare the run to the optimal prerecorded frames and report on each game for a complete summary of all the games. From there on the manual tester and the software development team will evaluate the results and report back to the team responsible for the continuous release of the service. This way the team will have a clear understanding of what has gone wrong in the code and will be able to find solutions fast for how to fix the errors.

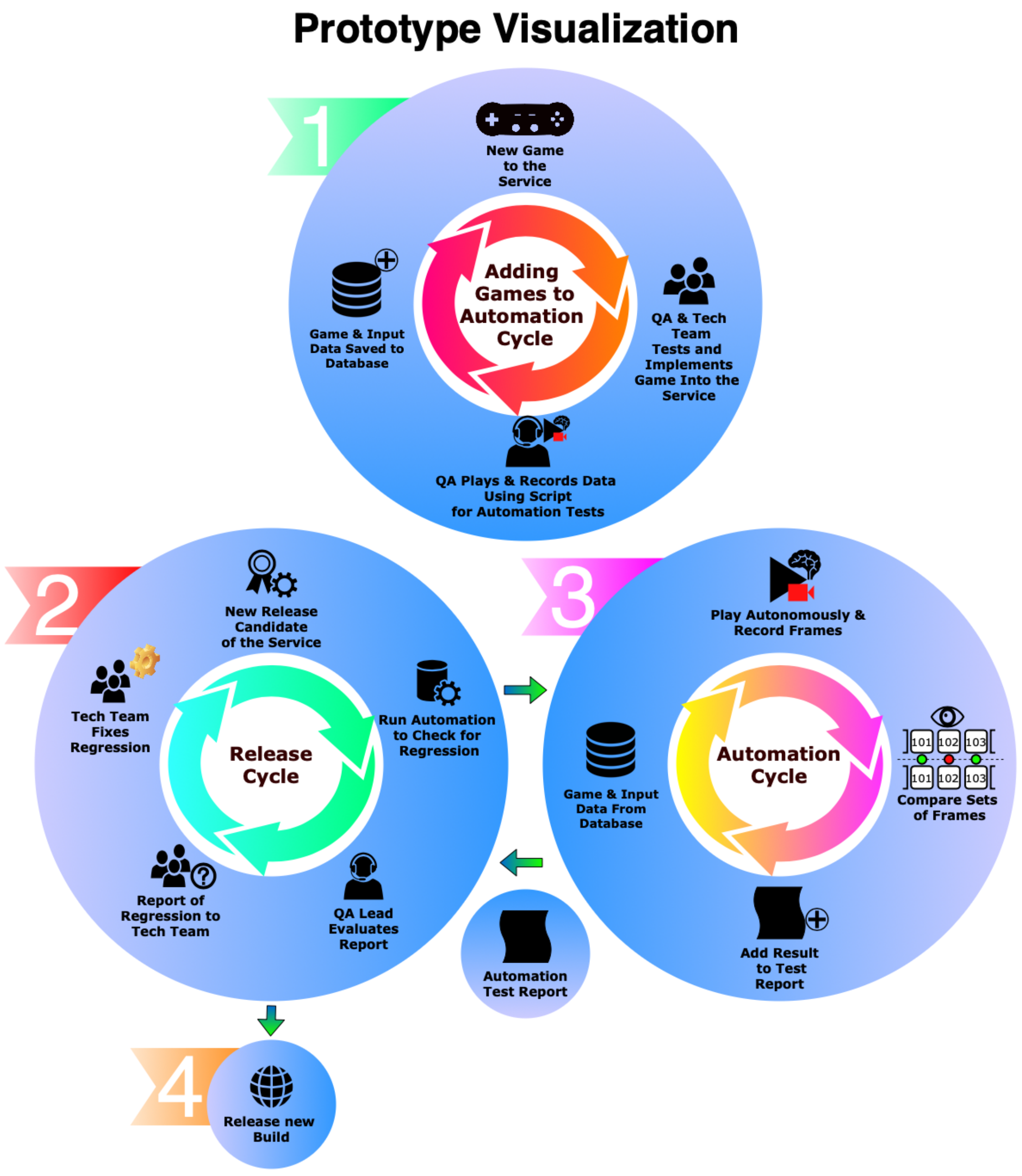
Prototype Visualization
1
New Game to the Service
Adding Games to Automation Cycle
QA & Tech Team Tests and Implements Game Into the Service
QA Plays & Records Data Using Script for Automation Tests
Game & Input Data Saved to Database
2
New Release Candidate of the Service
Release Cycle
Run Automation to Check for Regression
QA Lead Evaluates Report
Report of Regression to Tech Team
Tech Team Fixes Regression
3
Play Autonomously & Record Frames
Automation Cycle
Compare Sets of Frames
101 102 103
101 102 103
Add Result to Test Report
Game & Input Data From Database
Automation Test Report
4
Release new Build

Figure 4. Simple visualization of a possible release cycle and automation cycle after implementation of the prototype.

Figure 4 represents a possible test cycle after the implementation of the automation. Step one in the figure represents the recording of a manual session by the QA representative. Step two is a depiction of a new release of the service, which leads to step 3, the automation tests of each game on the service. Once the QA representative determines that the new version of the release is working as indented without any issues in the games, it can be made available to the customers. With this automation the speed and quality of tests result would be increased significantly, and costs of testing will cut down greatly.

3 Theoretical Background

First of all, to prove the probability of the functionality of the automation, we need tools that are best for prototyping purposes. Prototyping should be made simple and fast. The implementation should gather information on best practices of what works and what not. In agile development methodologies, we do not want to get stuck on one idea and try to make it work but instead explore several ones and choose the best one that best fits our needs. If a certain approach does not work for what is aimed for, we can quickly revert and try something else. This iteration-based approach is crucial in automated testing as well.

First of all, we want to ignore the existing codebase and the service entirely. The focus is put solely on testing out the idea of recording user input whilst playing a game natively on an Android device and then replaying those inputs. Natively means playing a game on the device by downloading it into the device memory.

This way we see if it is at all possible to make a computer play the games autonomously without a human physically interacting with the device. After this we want to gather each frame during the recording and replaying phases for comparison and to find out if regression has occurred. Finally, if all is successful, instead of playing the games natively, we start the games on the service’s local client on the device to see if latency has an effect on the results.

Latency is a delay in time between two events, one that is the cause the other one that is the effect of it. In other words, in mobile games, latency is the so-called lag, the time interval between doing an action and perceiving the result of it [4]. In streaming mobile games, it is essential to avoid latency as much as possible. Figure 5 gives a visual representation of latency.

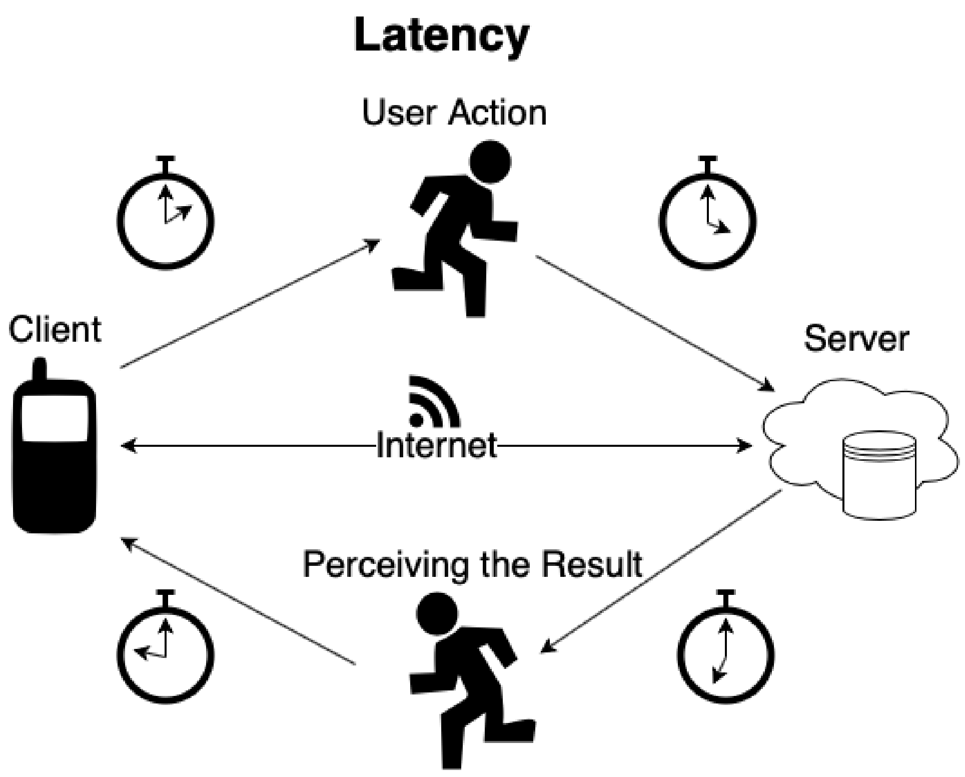

Figure 5. Visualization of latency. Time between user action from the client to server and receiving the result back on the client.

In this chapter we will discuss in-depth what tools were used and why, reasons why specific languages were chosen and the approach for the problem itself.

3.1 Python for Prototyping

To achieve simple and fast prototyping, Python was selected as the scripting language to explore the potential of the implementation idea. For this project this was also beneficial, due to the prior experience and knowledge of using this specific programming language. Python is a great language for scripting purposes and greatly used among automation tools in software engineering. Therefore, a great deal of research material can be found on this topic. With the help of libraries that Python provides, we can make quick changes and explore different approaches for this specific problem.

Python is an object-oriented programming language that also supports procedural and functional programming [5]. This is great for our purpose since it provides multiple ways to approach the execution. Python comes with various different libraries that we can benefit from. Examples of such benefits are: communicating with the computer's command-line, which is needed to interact with physical Android devices; reading and writing files into the computer for saving data necessary for the automation; comparing images for similarities and modifying images for better automation performance.

The initial step for the prototype is to explore the possibility of recording and repeating user inputs on Android devices. Python and its libraries are used for accessing the command-line tool which can make use of the Android Debug Bridge (adb) that allows communication with the physical Android device. Adb provides functionality for manipulating user inputs.

3.2 Interacting with Android using Android Debug Bridge

Adb is an Android command-line tool used for interacting with an Android device. Adb is a client-server program which contains three main components, a client, a daemon and a server. These three are used for sending commands on the device, running them and finally inspecting the outcomes of these actions. [6] These three components allow the user to install and launch games on the device, as well as test the functionality and get debugging information of recording and repeating input events for the automation. In addition, a video can be stored on the device for later use of comparing frames.

A client is the computer connected to the android device. In our case, the script will be sending commands to the client, who will then send these commands to the device connected to it via an USB-cable or a WiFi connection.

A daemon is a service that accepts and executes commands from a client. In this case the daemon is running on the computer and the android device simultaneously. On the device, daemon accepts and executes commands sent from our script. The client and the daemon are connected via the server which is a software running in the background sending the commands to the daemon. [6, 7]

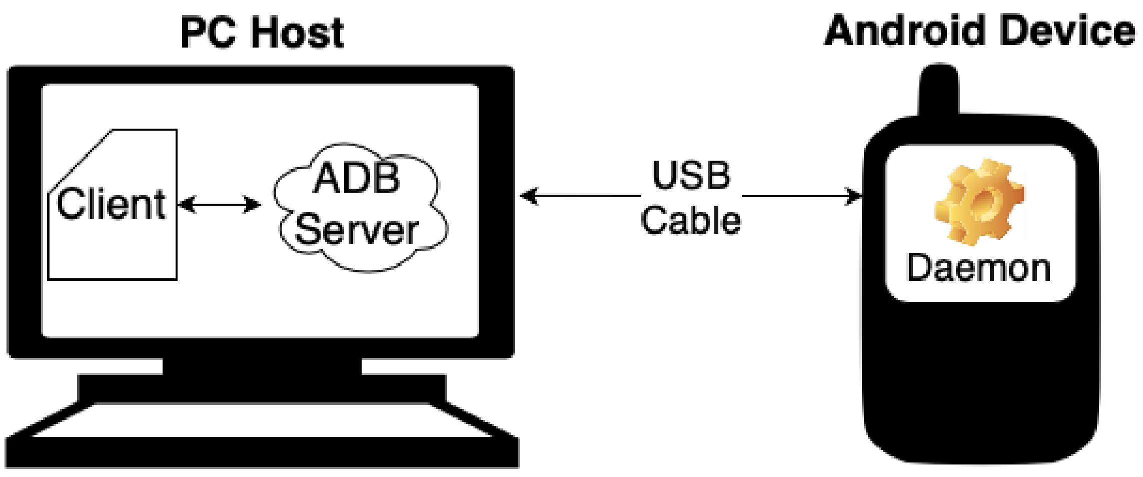

Figure 6. Visualization of adb client-server program.

A big challenge in the script is to understand how to record and replay input events. For this we need to understand the structure of Android's touch event data. On command-line this is done by using a command, adb *getevent* for getting information about input data and adb *sendevent* for sending input events to the device. In figure 6, this command would be sent from the client which then will be be processed by the android device daemon.

3.3 Understanding Types of Android Input Data Using Adb Getevent

When using an Android device, there are a number of things one can do regarding input events. The number is usually pretty much the same on all devices but can differ slightly from model to model. The most common input events a user is able to do on an Android mobile device are:

1. Tapping,
2. Swiping or holding,
3. Swiping with two fingers (i.e zooming in or out on a camera),
4. Up or down (mostly used for increasing or decreasing volume),

5. Power button,
6. Touch screen button (i.e fingerprint recognition)
7. Sound trigger (i.e activating voice assistant)

The information about the user's input events can be accessed using the Android's tool *Getevent*. *Getevent* runs on the Android device and provides two main data sets. The input devices supported by the Android device and input events those input devices use. [8] Input events also provide information on the range of its values. For example, the value on the device's horizontal plane might range from 0 to 255. For example, if one would press at the very left edge of the screen, the event should input a number of 0.

```
Terminal$ adb shell  getevent -lp
add device 1: /dev/input/event4
            name:     "hi3660_HI6403_CARD Headset Jack"
            events:
            KEY (0001): KEY_VOLUMEDOWN   KEY_VOLUMEUP   KEY_F14    KEY_MEDIA
KEY_VOICECOMMAND
            SW  (0005): SW_HEADPHONE_INSERT   SW_MICROPHONE_INSERT
            input props:
                        <none>
add device 2: /dev/input/event0
            name:     "soundtrigger_input_dev"
            events:
                        KEY (0001):
KEY_F14               KEY_F15               KEY_F16               KEY_F17
        KEY_F18
            input props:
                        <none>
add device 3: /dev/input/event3
            name:     "fingerprint"
            events:
                        KEY (0001): KEY_ENTER KEY_SEMICOLON
            KEY_GRAVE       KEY_LEFTSHIFT
                        KEY_Z
            KEY_V
            KEY_N                 KEY_SLASH
                        KEY_RIGHTSHIFT          KEY_KPASTERISK
            KEY_LEFTALT     KEY_UP
                        KEY_LEFT                KEY_RIGHT
            KEY_DOWN        KEY_INSERT
                        KEY_DELETE              KEY_MUTE
            KEY_KPEQUAL   KEY_KPPLUSMINUS
                        KEY_PAUSE               KEY_SCALE
            KEY_EXIT
            input props:
                        <none>
add device 4: /dev/input/event2
            name:     "hisi_on"
            events:
                        KEY (0001): KEY_POWER
            input props:
                        <none>
add device 5: /dev/input/event1
            name:     "hisi_gpio_key"
            events:
```

```
                        KEY (0001): KEY_VOL-
UMEDOWN        KEY_VOLUMEUP
            input props:
                        <none>
add device 6: /dev/input/event5
            name:     "huawei,ts_kit"
            events:
                        KEY (0001):  KEY_F1                BTN_TOOL_FIN-
GER       BTN_TOUCH
                        ABS (0003):
                        ABS_MT_TOUCH_MAJOR      : value 0, min 0, max 255,
fuzz 0, flat 0, resolution 0
                        ABS_MT_TOUCH_MINOR      : value 0, min 0, max 255,
fuzz 0, flat 0, resolution 0
                        ABS_MT_WIDTH_MAJOR      : value 0, min 0, max 100,
fuzz 0, flat 0, resolution 0
                        ABS_MT_WIDTH_MINOR      : value 0, min 0, max 100,
fuzz 0, flat 0, resolution 0
                        ABS_MT_ORIENTATION      : value 0, min 0, max 255,
fuzz 0, flat 0, resolution 0
                        ABS_MT_POSITION_X                   : value 0, min 0,
max 1440, fuzz 0, flat 0, resolution 0
                        ABS_MT_POSITION_Y                   : value 0, min 0,
max 2560, fuzz 0, flat 0, resolution 0
                        ABS_MT_TRACKING_ID      : value 0, min 0, max 15, fuzz
0, flat 0, resolution 0
                        ABS_MT_PRESSURE         : value 0, min 0, max 1080,
fuzz 0, flat 0, resolution 0
            input props:
                        INPUT_PROP_DIRECT
```

Listing 1. Information about Android device's input devices and input events using *adb getevent*.

Here is shown an example output of the adb *getevent* tool. This terminal command lists information about all the input devices and their events. For this specific device used in the previously illustrated log file, there are six different input devices. These are:

1. Device 1: Headset jack
2. Device 2: Sound trigger input dev
3. Device 3: Fingerprint button
4. Device 4: Power button
5. Device 5: Volume up and down
6. Device 6: Touch screen

One goal of the prototype is to explore recorded and repeated user inputs while playing a game. Therefore, the device that represents touch screen events has to be used, since that is what the user will be operating with while playing. For our specific Android device, it would be input device number 6 from the *getevent* tool report. Before recording this input device and its events, it is important to understand what they mean.

```
add device 6: /dev/input/event5
```

First line of the touch screen device informs the number of the device, 6, which in this case is not that important. What comes after that is important for this use case, since it tells us which event number within the Android device we are communicating with. This will be used to record and in the future replay touch screen input events. For this device that is ”/dev/input/event5”. Android OS is a modified version of Linux operating system and therefore uses the Linux input system. This line represents a generic linux input event interface. [9]

```
name:     "huawei,ts_kit"
```

Name presents the name of the event. This name is created by the company that created this specific Android device operating system.

```
KEY (0001):  KEY_F1     BTN_TOOL_FINGER         BTN_TOUCH
```

After the name the events are presented with their properties. Key events represent a single key that has the value of 0 or 1. When this key is activated, an event with value 1 with the key's code is emitted. Once the key is released, an event with the value of 0 is emitted. Some keys have special meanings. On this device these are *BTN_TOOL_FINGER* and *BTN_TOUCH*. When the touch screen of the device is used a value of 1 is emitted by the *BTN_TOOL_FINGER* key. The *BTN_TOOL_<name>* key is used with touchscreen, trackpads and tablets. A device with a pen, would likely use a *BTN_TOOL_PEN* key when using the pen. This device only uses a finger and therefore it is called *BTN_TOOL_FINGER*. The *BTN_TOOL* is set to 1 when there is meaningful contact on the device. For example, this means that when using a pen on a device, it might not have to physically touch the screen in order for the key to be active. In these cases, touchscreen devices also implement the *BTN_TOUCH* key. This key represents the action of physically touching the screen. Once the screen makes contact with a finger or anything else, a value of 1 is emitted. *BTN_TOOL* might emit 1 but does not mean that *BTN_TOUCH* is active. [10]

```
ABS (0003): ABS_MT_TOUCH_MAJOR      : value 0, min 0, max 255, fuzz 0, flat 0,
resolution 0
                        ABS_MT_TOUCH_MINOR      : value 0, min 0, max 255,
fuzz 0, flat 0, resolution 0
                        ABS_MT_WIDTH_MAJOR      : value 0, min 0, max 100,
fuzz 0, flat 0, resolution 0
                        ABS_MT_WIDTH_MINOR      : value 0, min 0, max 100,
fuzz 0, flat 0, resolution 0
                        ABS_MT_ORIENTATION      : value 0, min 0, max 255,
fuzz 0, flat 0, resolution 0
                        ABS_MT_POSITION_X                   : value 0, min 0,
max 1440, fuzz 0, flat 0, resolution 0
                        ABS_MT_POSITION_Y                   : value 0, min 0,
max 2560, fuzz 0, flat 0, resolution 0
                        ABS_MT_TRACKING_ID      : value 0, min 0, max 15, fuzz
0, flat 0, resolution 0
                        ABS_MT_PRESSURE         : value 0, min 0, max 1080,
fuzz 0, flat 0, resolution 0
```

Listing 2. Touchscreen input device information.

Touchscreen devices have multiple touchscreen event properties. These are represented by *ABS* events. *ABS_MT_<name>* represents a multitouch property of an event. For example, when touching a screen with two fingers, a multitouch event is happening. On this device all the touchscreen events support a multitouch property, meaning the user is not limited to a single action as it was when touchscreen devices first came on the market. *ABS* events describe absolute changes in these properties, for example when touching the screen, the event emits an absolute value of the coordinates of the touch location. [10]

There are nine multitouch properties on the current device which each have their own functionality. The minimum number of multitouch properties on a touchscreen device is 2, those being *ABS_MT_POSITION_X* and *ABS_MT_POSITION_Y*. These allow multiple physical contacts on the screen to be tracked. Here a list is provided on the functionality of each property:

1. *ABS_MT_TOUCH_MAJOR*: Major axis contact length in surface units
2. *ABS_MT_TOUCH_MINOR*: Minor axis contact length in surface units
3. *ABS_MT_WIDTH_MAJOR*: Major axis contact length of the approaching tool in surface units
4. *ABS_MT_WIDTH_MINOR*: Minor axis contact length of the approaching tool in surface units
5. *ABS_MT_ORIENTATION*: Orientation of the contact area
6. *ABS_MT_POSITION_X*: Contact x coordinate

7. *ABS_MT_POSITION_Y*: Contact y coordinate
8. *ABS_MT_TRACKING_ID*: Unique ID for a given input during its lifecycle
9. *ABS_MT_PRESSURE*: The pressure of contact area

[11]

By recording these properties and using them for repeating, autonomous play should be possible to achieve.

3.4 Understanding Android Input Stream Using Adb Getevent

The adb *getevent* tool can give access to a live input stream while using the device. This then can be recorded into a text file which can later be used to replay the same events. There are many options we can use the getevent tool in order to get the necessary data we want. On the command-line tool we can see these options using *adb shell getevent -help*.

```
Terminal$ adb shell getevent -help
Usage: getevent [-t] [-n] [-s switchmask] [-S] [-v [mask]] [-d] [-p] [-i] [-l]
[-q] [-c count] [-r] [device]
    -t:     show time stamps
    -n:     don't print newlines
    -s:     print switch states for given bits
    -S:     print all switch states
    -v:     verbosity mask (errs=1, dev=2, name=4, info=8, vers=16, pos.
events=32, props=64)
    -d:     show HID descriptor, if available
    -p:     show possible events (errs, dev, name, pos. events)
    -i:     show all device info and possible events
    -l:     label event types and names in plain text
    -q:     quiet (clear verbosity mask)
    -c:     print given number of events then exit
    -r:     print rate events are received
```

Listing 3. Output of adb “shell getevent -help” command.

To inspect how the previous chapters devices and events are outputted when dumping input stream, the option “-l” can be used. Calling “adb shell *getevent* -l”, will output live data on the terminal. In the following console output is shown how these are presented.

```
/dev/input/event5: EV_KEY       BTN_TOUCH            UP
/dev/input/event5: EV_SYN       SYN_REPORT           00000000
/dev/input/event5: EV_SYN       SYN_MT_REPORT        00000000
/dev/input/event5: EV_ABS       ABS_MT_PRESSURE      000000ba
```

```
/dev/input/event5: EV_ABS       ABS_MT_POSITION_X    00000285
/dev/input/event5: EV_ABS       ABS_MT_POSITION_Y    0000075a
/dev/input/event5: EV_ABS       ABS_MT_TRACKING_ID   00000000
/dev/input/event5: EV_ABS       ABS_MT_WIDTH_MAJOR   00000000
/dev/input/event5: EV_ABS       ABS_MT_WIDTH_MINOR   00000000
/dev/input/event5: EV_SYN       SYN_MT_REPORT        00000000
/dev/input/event5: EV_KEY       BTN_TOUCH            DOWN
/dev/input/event5: EV_SYN       SYN_REPORT           00000000
/dev/input/event5: EV_SYN       SYN_MT_REPORT        00000000
/dev/input/event5: EV_KEY       BTN_TOUCH            UP
/dev/input/event1: EV_KEY       KEY_VOLUMEDOWN       DOWN
/dev/input/event1: EV_SYN       SYN_REPORT           00000000
/dev/input/event1: EV_KEY       KEY_VOLUMEDOWN       UP
/dev/input/event4: EV_SW        SW_HEADPHONE_INSERT  00000001
/dev/input/event4: EV_SW        SW_MICROPHONE_INSERT 00000001
```

Listing 4. Example output of “*adb shell getevent -l*” command on the terminal.

During this small recording, the device screen was touched, volume was increased and decreased and finally an audio cable was inserted. These three events are categorized as the event numbers 5, 1 and 4. The “EV_SYN” is used to separate events. By separating an event, the device knows an event has finished and a new one can be detected. All this data can be packed into a struct layout representation in the following manner:

```
struct input_event {
            struct timeval time;
            unsigned short type;
            unsigned short code;
            unsigned int value;
};
```

Listing 5. Input layout representation in struct. [12]

‘Time’ is a representation of when an event occurred, also known as the timestamp. Time can be viewed by adding the option of “-t” to the *getevent* call. Type is the key (EV_KEY), touch screen event (EV_ABS) or event separator (EV_SYN). The code tells what property of the key occurred and finally value gives the information about the value of the event. [12] For example, the first line, type tells us a key was pressed and the code says that the touchscreen made contact with the finger and value says that the finger was lifted off the screen. Since the automation does not need to read the events in plain text, the option “-l” can be removed. This will make the input events be represented in a hexadecimal number. Processing this is much faster for the computer. For the human eye, the labels were used to demonstrate and understand the structure of the input stream. After removing the label and replacing it with timestamp option “-t”, the output looks the following:

```
[   62605.659306] /dev/input/event5: 0003 0035 000001f7
[   62605.659306] /dev/input/event5: 0003 0036 000006ee
[   62605.659306] /dev/input/event5: 0003 0039 00000001
```

Listing 6. Example output of "*adb getevent -t*" command on the terminal.

Since mobile games are in most cases only played using the touchscreen and not with any of the other input devices in a mobile gadget. It is safe to assume, at least for the prototype, that for recording and repeating touch events, only touchscreen events are needed. The recording log should contain the timestamp, type, code and value but the "/dev/input/event5" is not needed. In that case, the event number still needs to be stored somewhere in order to communicate with the correct device when repeating the touchscreen events. Here *getevent* can be called and the data can be stored in the following manner:

```
Terminal$ adb shell getevent -t /dev/input/event5
[   69182.744623] 0003 003a 00000087
[   69182.744623] 0003 0035 000000d4
[   69182.744623] 0003 0036 00000738
```

Listing 7. Example output of getevent with addition of the input device.

That is the data format to be stored into a log, which can later be repeated. If timestamp was not used and the events were replayed, there would be no delay between these input events and the functionality the prototype aims for would not be reached. Using the timestamp, the time between each event can be calculated for a realistic repetition of input events.

3.5 Understanding Adb Sendevent

Now that recording of user actions on an Android device have been achieved, it is time to replay those same touch events. Adb provides another tool called *sendevent*. *Sendevent* is a linux input event same as the *getevent*. Using *sendevent*, it is possible to interact with the input devices on the mobile device. The *sendevent* tool requires four arguments to function. These being the input device to send an event to, type, code and value. Same as on the *getevent* input event layout. Whereas *getevent* does not require the specification of the input device, *sendevent* does since it is communicating straight to a specific input device. This is why it is important to store the device event number.

Sendevent does not use timestamps, therefore, to simulate pauses between inputs, external methods are required.

Getevent tool returns the type, code and value in hexadecimal numbers, but *sendevent* requires them to be in decimal numbers. This means these three values must be converted before being able to use *sendevent*. Python will be used for this purpose in the later phases. It is also important to remove the semicolon appearing after the event number. Here is an example of calling *sendevent* to press the power button:

```
adb shell sendevent /dev/input/event2 1 116 1
adb shell sendevent /dev/input/event2 0 0 0
adb shell sendevent /dev/input/event2 1 116 0
```

Listing 8. Example command calls for pressing the device power button using *adb sendevent*.

This requires three *sendevent* command calls: the simulate power button press down, the event separator and finally the simulate power button press up. Same applies for touchscreen input events.

```
struct input_event {
            struct timeval time;
            __u16 type;
            __u16 code;
            __s32 value;
};


int sendevent_main(int argc, char *argv[])
{
            int fd;
            ssize_t ret;
            int version;
            struct input_event event;

            if(argc != 5) {
                        fprintf(stderr, "use: %s device type code value\n",
                        argv[0]);
                        return 1;
            }

            fd = open(argv[1], O_RDWR);
            if(fd < 0) {
                        fprintf(stderr, "could not open %s, %s\n",
                        argv[optind], strerror(errno));
                        return 1;
            }
            if (ioctl(fd, EVIOCGVERSION, &version)) {
                        fprintf(stderr, "could not get driver version for %s,
                        %s\n", argv[optind], strerror(errno));
                        return 1;
            }
            memset(&event, 0, sizeof(event));
            event.type = atoi(argv[2]);
            event.code = atoi(argv[3]);
            event.value = atoi(argv[4]);
            ret = write(fd, &event, sizeof(event));
            if(ret < (ssize_t) sizeof(event)) {
                        fprintf(stderr, "write event failed, %s\n", strer-
                        ror(errno));
                        return -1;
            }

            return 0;
}
```

Listing 9. Android's internal sendevent function for repeating touch events. [13]

The above listing showcases the sendevent file within the android device which takes in the input event and then writes it, therefore simulating a touch event.

3.6 Accessing Command-Line Using Python Module, Subprocess

The Python prototype script requires access to the android device connected to a computer, in order to download and start games, record and replay touch events and frames. This access can be retrieved by using a Python module named *subprocess*. *Subprocess* allows running applications or programs by creating new processes. This way access can be achieved to the command-line through the Python script. For obtaining the results of the command-line instructions, *subprocess* provides input/output/error pipes and exit codes on the commands called. With pipes, the script can send information to the process to end it and then read the result. For example, once the replaying of the touch events is done, the script reads the output of the process for further inspection. Another example, when starting a game on the device, the *subprocess* call returns the result if starting the game was successful. [14]

Based on the functionality needs, the script uses two *subprocess* functions to execute adb commands. The two functions are *subprocess.Popen* and *subprocess.call*. *Popen* can be used for example when recording the touchscreen input events. The reason behind this is that each time an event occurs, the script needs to be notified so that the event can be stored for later use. The *subprocess.call* function of *subprocess* can be used for example when pushing an APK in the Android device and when replaying input events using *sendevent*. With *subprocess.call* the information is only then needed if the command failed. The *call* function waits for the event to happen and then the script continues with its next executions. [14]

3.7 Recording and Comparing Frames in Python

Frame rate is a big factor in games and can have an effect on the functionality of the automation prototype. Frame rate tells how many consecutive images appear on the screen within a second. Frame rate is also known as frames per second (fps). Fps can differ each time a user plays a game. Playing a game on a different device can also have an effect on the fps. During recording and replaying of touch events, the goal is also to record frames. Since the fps can differ each playthrough, the number of frames can differ from session played by the manual tester and the session ran autonomously. Video recording of the session can be used to try to minimize this problem. If instead of video, frames were recorded, the performance could be diminished, and the number of frames recorded would be determined by the fps that the game ran in.

Adb provides a screen recording tool for recording videos on the device. Adb screenrecord has a fixed fps when recording, therefore minimizing the error of fluctuating fps. Once the recording of touch events has finished, the video recording will be finished as well. Android stores this screen recording within the device, which can then be pulled into the computer.

The video file needs to be processed into a set of frames. If the playthrough lasted a minute and the video fps was 25, that leads to 60 seconds times 25 fps which equals to 1500 frames. Due to the same video length and fps in both recording and replaying, the number of frames should be equal. Within python, this video can be chopped into frames through a library called OpenCV.

OpenCV is an open source software library that specializes in computer vision and machine learning. OpenCV provides functions for modifying image data and for comparing two images in similarity. OpenCV can also be used for modifying images by lowering their quality for better overall software performance. [15]

Once a game has been manually and autonomously played and the frames have been captured in both sessions, the two sets of frames need to be compared. With comparing frames, the script looks for differences between the frames. These differences aim to point out issues in the game after a new release of the service. There are multiple different libraries for comparing images and for the prototype a library called *diffimg* is used. Each frame from both sets at the same position will be compared to each other using diffimg and the returned result of similarity in percentages will be used to determine if regression has occurred.

Diffimg takes two images as input parameters with the same size and same color channels. This is important since diffimg compares each pixel from the same location on both images by their color channel. If regression has occurred in the autonomously played game this library should be able to say if something has broken by comparing the color channels. [16] When comparing images, if the difference between the two is larger than a predefined percentage, those frames will be flagged and added to a test report. Diffimg can also provide the report with a generated image showing the difference between the compared images. The flagged images will be then reviewed by the quality assurance team representative. [16]

4 Practical Implementation with Python

The following chapter focuses on the actual implementation of the python script for the automation prototype. All-important functionalities mentioned in the previous chapter will be explained in detail using examples of the code.

4.1 Installing and Getting the Game Started on the Device

All the games on the service are stored in a database and in order to create test data for the automation for a specific game, the game needs to be downloaded on the device and started up for recording. Games on Android devices come in the format of an android application package (apk). Some games need more space than others and therefore need to include one or more expansion packages called OBB files. To install the apk and obb files, it is safe to first delete the existing files of the game for a fresh state of the game when it is started. By having a newly created game state, everytime when recording or replaying happens, it is made sure that each session should be the same. The game restarts from the beginning and simulates the actions as if it was the first time it was launched. Starting a game on the android device requires the information of the package and activity name of the apk being downloaded. Using the package name, the adb can detect the files on the device to delete them. Activity name can be stored and later used for starting the game.

```
def getPackageAndActivityName(self, apk):
        # Get game package name using android asset packaging tool
        exit_code, result = self.shell('aapt', 'dump', 'badging', apk)
        if exit_code:
            elog('Failed to capture package name. Check if APK file path is 
correct.')
            sys.exit()

        # Get package name from output using regex and store for later use
        apk_package_names = re.finditer(r'package: name=\'(\S+)\'.*', re-
sult)
        for i in apk_package_names:
            self.package_name = i.group(1)

        # Get launchable activity name using regex and store for later use
        apk_activity_name = re.finditer(r'launchable-activity: 
name=\'(\S+)\'.*', result)
        for i in apk_activity_name:
            self.activity_name = i.group(1)
```

Listing 10. Function for getting package and activity name of the apk file.

The function getPackageAndActivityName takes as parameters a given apk file. For getting the package and activity name, an android asset packaging tool called aapt is used. [17] To get the package and activity name from the terminal output a regular expression module is used. Regular expression searches pieces of strings from text by using specified search arguments.

```
def downloadApkObbOnDevice(self, apk, obb=None):
        ## Download APK on connected device
        ilog('Downloading APK to device...')
        exit_code, result = self.shell('adb', 'uninstall', self.pack-
age_name)
        exit_code, result = self.shell('adb', 'install', '-r', apk)

        # Download OBB file to device if included
        if obb:
            # Create folder for OBB file with the name of the package name
            ilog('Creating OBB folder to device via adb...')
            exit_code, result = self.shell(
                                             'adb',
                                             'shell',
                                             'mkdir',
                                             '/sdcard/Android/obb/' + self.pack-
age_name)

            # Push OBB file/s to OBB folder
            for obbFile in obb:
                ilog('Pushing OBB...')
                # Push OBB file to folder
                exit_code, result = self.shell(
                                                 'adb',
                                                 'push',
                                                  obbFile,
                                                 '/sdcard/Android/obb/' +
self.package_name + '/')
                if exit_code:
                    dlog(result)
                    elog('Failed to push OBB. Check if OBB file path is cor-
rect.')
                    sys.exit()
```

Listing 11. Function for downloading apk and obb on the connected android device.

The above function is used for downloading the apk and obb files from the pc to the connected device that is being used for testing. Adb is used for downloading and pushing files to the device. Obb files need to be pushed to a specific location in the device's memory. This location in the memory is ‘/sdcard/Android/obb/’. A folder for this specific obb file needs to be created with the package name of the apk in order for the game to work.

```
def startApkOnDevice(self):
        # Start APK on connected device
        ilog('Launching APK on device...')
        exit_code, result = self.shell(
                            'adb',
                            'shell',
                            'am',
                            'start',
                            '-n',
                            os.path.join(self.package_name, self.activity_name))
        if exit_code:
           elog('Starting APK failed')
           sys.exit()
```

Listing 12. Function for starting apk on the android device.

Starting an apk requires both the apk package name and the activity name. In this function adb start command was used.

```
def cmdlineRepr(self, *args):
    return ' '.join(shlex.quote(arg) for arg in args)

def shell(self, *args, echo_type=False, parallel_task=False, cwd=None):
    log_txt = self.cmdline_repr(*args) + '\n'
    if echo_type:
        dlog(self.cmdline_repr(*args))

    proc = subprocess.Popen(
                          args,
                          stderr=subprocess.STDOUT,
                          stdout=subprocess.PIPE,
                          encoding='UTF-8',
                          cwd=cwd
                          )
    if parallel_task:
        return proc

    while proc.poll() is None:
        try:
            for line in proc.stdout:
                log_txt += line
                if echo_type:
                    dlog(line)
            result = proc.wait()
        except KeyboardInterrupt:
            break

    result = proc.wait()
    proc.terminate()
    return result, log_txt
```

Listing 13. General function for using subprocess.

All of the previous functions used the *self.shell* method which was created for accessing the command-line using the *subprocess module*. This function is presented above. The *shell* function takes in the argument of **args* which is the actual command-line function call. The parameter *echo_type* is used for debugging purposes of logging the results on the command-line. This is for better visualization of what is happening. *Parallel_task* parameter is a boolean value which if true exits the code after calling the *subprocess* module. This is if another *subprocess* process wants to be started at the same time. *Cwd* specifies the working directory to another before executing the process. This is used later on when implementing the script with the existing codebase.

4.2 Recording and Replaying Touch Events

After downloading the game and starting it on the device the script will also start recording the video and touch events. Touch events are recorded using adb *getevent* and instead of using the timestamp provided by the tool itself, the timestamp is captured manually. These touch events are stored into an array that can be later stored into a file.

```
def recordTouchEvents(self):
        inputEvents = []

        # Start time for capturing timestamp for touch events
        startTime = int(round(time.time() * 1000))

        geteventCall = subprocess.Popen(self.adb_shell_command + [b'gete-
vent', self.touchscreen_device], stdout=PIPE)
        while geteventCall.poll() is None:
            try:
                inputEvent = adb.stdout.readline().strip()
                currentTime = int(round(time.time() * 1000))
                timestamp = currentTime - startTime
                inputEvents.append("%s %s\n" % (timestamp, input_event))
            except KeyboardInterrupt:
                currentTime = int(round(time.time() * 1000))
                timestamp = currentTime - startTime
                inputEvents.[:-1] + ("%s %s\n" % (timestamp, input_event))
                geteventCall.terminate()
                break

        return inputEvents
```

Listing 14. Function for recording user touch events while playing a game.

In this function the variable startTime is used to start counting the time immediately once the game starts. This is because *getevent* only starts the recording once the first input is captured. When repeating the inputs, it is important that the delay between starting the game and the first input is also taken into account. The person recording the manual game session can decide how long the automation run can be by interrupting the recording once they feel necessary. This interruption can be done using the keyboard while on the command line. When the keyboard interruption occurs, the *getevent* process is terminated so that it does not continue in the background.

```
def replayTouchEvents(self, userInputs):
        ilog('Replay started...')
        startTime = int(round(time.time() * 1000))

        for inputEvent in userInputs:
            inputData = STORE_LINE_RE.match(inputEvent.strip())
            timeStamp, etype, ecode, data = inputData.groups()

            currentTime = int(round(time.time() * 1000))
            timeGone = currentTime - startTime

            # Simulate delay between touch events using a loop
            while int(timeGone) < int(timeStamp):
                currentTime = int(round(time.time() * 1000))
                timeGone = currentTime - startTime
                if timeGone > timeStamp:
                    break

            # Send touchscreen input event to device
            sendeventCall = self.adb_shell_command + [b'sendevent',
self.touchscreen_device, etype, ecode, data]
            if subprocess.call(sendeventCall) != 0:
                raise OSError('sendevent failed')

        ilog('End replaying')
```

Listing 15. Function for replaying user input events for autonomous play.

Once recording of inputs is done the QA representative can run the script for autonomous play. Here the game is re-downloaded and the recorded inputs of the manual play session are repeated. This way the game is played by the computer. The above function handles repeating of the inputs by taking in an array of inputs as a parameter, reading its data, looping to simulate delay between inputs and sending the input to the device using the adb sendevent call. Once all the inputs have been run, the autonomous play is finished.

4.3 Capturing Test Session Frames

Capturing video is done during the recording and replaying of touch events. Capturing frames is done by recording a video of the play sessions and then later on splittin that video into a set of frames.

```
def recordVideo(self):
        screenrecordCall = self.shell(
                                        'adb',
                                        'shell',
                                        'screenrecord',
                                        '/sdcard/' + self.fileName)
        while screenrecordCall.poll() is None:
            try:
            except KeyboardInterrupt:
                screenrecordCall.terminate()
                break
```

Listing 16. Recording video while playing using adb's screenrecord function.

Recording of video is done in a similar manner as with recording input events. An adb process is started using the subprocess module and the recording is finished once the tester interrupts it using the pc keyboard. When that happens, both the video and touch event recording are ended at the same time.

```
def pullVideoAndChopToFrames(self, fileName, folderLocation, frameFolder):
        # Pull video from device
        if subprocess.call(self.adb, 'pull', '/sdcard/' + fileName + ' ' +
folderToSaveTo + '/') != 0:
            raise OSError('Pulling screenrecord from device failed')

        ilog('Capturing frames... (This may take several minutes)')

        # Capture frames from video for image comparison
        vidcap = cv2.VideoCapture(fileName)
        success,image = vidcap.read()
        count = 0
        while success:
          cv2.imwrite(frameFolder + "/frame%d.jpg" % count, image)     #
save frame as JPEG file
          success,image = vidcap.read()
          count += 1

        ilog('Frames captured: ' + str(count))
        ilog('Recording finished')
```

Listing 17. Function for getting frames out of a video.

After recording for both manual and autonomous play, the video file can be found in the device storage. This video can be pulled from the device to the computer for it to be chopped into a set of frames and stored on the computer's hard drive. These frames along with the touch events can then later on be stored onto a database. Capturing frames is done by the above function using the OpenCV open source library. By reading the video, it is able to go through each frame and that frame can be then stored. The function also prints out the number of frames captured on the command-line tool.

4.4 Comparing Sets of Frames

Now that a game has been played manually, autonomy and each frame from both play sessions have been captured, these two sets of frames can be compared to each other. If all the frames at same positions are the same the both play sessions should be equal, meaning same touch actions have happened and same results have occurred within the game.

```
def compareSetsOfFrames(self, frameSet01, frameSet02):
        for frame1, frame2 in zip(frameSet01, frameSet02):
            # Percentage difference between two frames
            frameDifference = diff(frame1, frame2)
            # If difference is greater than 0.2 capture the frames and its
difference
            if frameDifference > self.frameDifferanceThreshold:
                self.addToTestReport(frame1, frame2, frameDifferance)
```

Listing 18. Function for comparing two sets of frames to each other.

The above function is used for comparing each frame and if the difference between those two frames is larger than a predefined percentage that the qa representative has specified, the frames are determined different and added to the test report. This test report is then later evaluated by the qa representative.

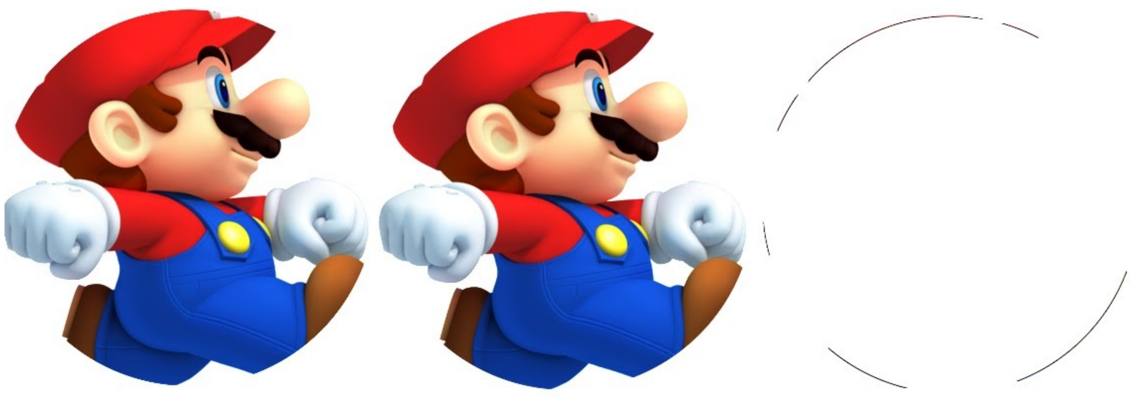

Figure 7. Example of two images compared and their result using diffimg. The first two images differ by 0.731961813597% [16].

Figure 7 represents two images being compared by diffimg. Here the two characters are the two frames being compared. The right most images is the difference between the two characters. This is similar what will be outputted into the test report provided to the QA team.

4.5 Implementing a Cloud Service Client to the Script

Previously recording and replaying of a session was done using a native apk, meaning that the game was not running on the service, but it was running like a regular game that a user would download. Since the goal is to test for regression on the games running in the cloud, the same methods were used but instead of starting a game on the device using adb, a client of the service was started on the device with a specified game. This way the script would actually get results on the service’s games instead. Bringing in the client the process is the same but new variables come into play. When games are played on the cloud, latency comes along, which can affect the results of replaying touch events and comparing images.

5 Results and Future Improvements

This prototype was implemented with the scope of testing the functionality of a possible future rigorous regression testing for a gaming service. Therefore, the evaluation of the results, outcomes and future improvements of this implemented script, has to be done under the consideration of the limitations a prototype has. The following section examines the results of implementing this prototype in Python and outlines possible future developments that could enhance the automation testing process.

5.1 Results

The overall idea of the prototype was proved to work but would still require work and enhancing. Downloading and starting games worked as intended. Getting user touch data and recording a video functioned as intended. Repeating touch events using sendevent, showed to be a bit too slow for its purpose. This was due to the constant function call of sendevent. It did work well enough to give a good demo of what it could be.

Another problem with recording touch data using *getevent* is that not all android devices support the same input data devices and device events. Meaning, if the manual test run recording was done using device A, repeating the inputs on device B might not work at all because device B input events do not match to one of A’s. Therefore, the touch data would need to be converted to a global touch data format that could be used on all devices. Another problem with using different devices for recording and repeating was that the device screen resolution might be different. This will also play a factor when repeating touch events. This would have to be taken into account when converting the data to a global format.

Mobile games and applications require testing on multiple devices to find out if specific devices function differently with the game or application. This is why it is important to be able to test for regression as well in this case on multiple devices. Regression might only occur on a set of specific devices, therefore as many devices that could be tested on, the better. Running an automation on multiple devices, will prolong the duration but that is what automation is for.

Resolution problems also occur when comparing frames. If the device used for recording frames during the manual test recording and another device used for the automation have different screen resolutions the recorded frames will also be in different sizes. Module that was used for comparing images requires the two images be the same size and have the same color format. Frames should then be converted into the same resolution in order to be compared. Android device screens can also differ in the way colors are shown which can also lead to problems.

Adb screen record which was used for recording video of the connected device screen is not supported on all android devices, meaning another way of recording these images would be required. There are many python libraries that provide this functionality, but performance is important while recording, therefore a performant way of recording should be prioritized.

Early on one of the biggest challenges acknowledged was the fact that regression can occur at any point of the game. The prototype aims to solve this problem by autonomously testing the game as far as the tester intends, meaning that that could be even to the end of the game. This would mean that tens of thousands or even more frames would be captured during the test runs. These frames would require large databases in where to be stored but the biggest issue happens when comparing the frames to the manual test session run. Comparing for example two sets of frames with each having over 20 000 images in them can take a really long time. Here it might be required to find a better solution of finding out regression in these games or to just be satisfied with the automation playing for example a few minutes.

Method of comparing each frame to the corresponding frame in the other set does not always work, since the play sessions might not be matched exactly timewise. For example, frame number 101 from set A could appear at frame number 105 in the set B. Here the comparison would compare A and B's frame 101 and it would think there is regression since they are different. With more frames the issue becomes more and more prevalent. As the automation is intended to run on a client of the cloud service, latency is introduced to the games. Latency enlarges this issue of frames occurring when they are supposed to.

Latency brings a big factor when running the automation, since games might load slower, inputs might be taken in milliseconds later. In short test runs this is not that big of a problem but with longer sessions this can become more of a problem. Recording and repeating test sessions should be done in an optimal as possible environment, meaning that latency would be at a minimum. When recording the test run this should also be taken into account by playing the games a bit slower and waiting between each input. This way when the game is run autonomously, the delays between inputs allow the game to load a bit longer.

Latency can also bring buffering in games. Here some of the frames recorded might be slightly blurred. Since this blurring does not necessarily occur in all test runs of the game, the comparison might think that blurring is regression. The automation should be some-how taught to understand when blurring is occurring and when not and more importantly when blurring is a result of regression and when not.

Randomization in games brings in challenges that the automation currently does not handle. A game can behave differently on each session leading to the touch events to not work since what was there before no longer exists. These types of games will not work with this type of automation. Same challenge happens with multiplayer games where actions are not determined by the player and the game but the surrounding envi-ronment within the game.

A lot of issues were unknown, and many issues were ignored in the beginning of this prototype. This was intentional since the goal was to find if this way of automating was possible at all. Even though the test results proved to not be the most reliable, the idea behind it seemed to work as intended. The project brought out issues and ideas that were not thought of before. With possible enchantments this type of automation definitely has potential.

5.2 Improving Recording and Replaying

During the development of the prototype many ways of improving it were trialed upon, like increasing performance by modifying images, modifying touch event data for compatibility with different devices, modifying android’s input system to receive multiple touch events at once along with many other things.

One of the biggest problems of the prototype was the use of adb’s sendevent command. For this purpose, the commands performance was too slow. One solution that was given an opportunity was modifying the sendevent command itself within the device. Instead of calling the command for each input event, the sendevent file was modified so that it would take all the input events and loop through them within itself. This was done by adding this implementation to the method itself and then replacing it with the existing sendevent c file within the device. This newly modified file would also need to be made compatible with different device architecture types like arm and x86 that android devices use.

Calculating latency for each input could solve the issue of inputs being repeated when they should not be. Each time an input is saved and replayed; the latency could be calculated so that the script could simulate similar latency during repetition of inputs of autonomous play.

Another angle towards this problem was created. Here playing, recording of inputs and frames and autonomous play were done on a pc client of the service. This way the implementation could be adapted to existing codebase and many of the existing issues with the prototype were solved. Touch data was globalized, resolution size was the same every time for touch events as well as recorded frames, repeating touch events and recording frames was also performan. Using the automation on a pc instead of a physical device takes away the possibility of testing on multiple different physical devices.

5.3 Machine Learning and Audio Processing for Finding Regression

Comparing every frame to each other can be slow and insufficient, especially when there are thousands or even tens of thousands of frames. That is why better automation options should be looked into. Machine learning and neural networks can play a key role in the solution for this issue. In order to find regression using visual imagery, deep learning could be a viable option. Especially convolutional neural networks (CNN) which are a class of deep neural networks. This method could be used for analyzing visual imagery which is exactly what this prototype aims to do. CNN can also be used for video recognition. [18]

Idea here is to teach CNN to recognize regression by feeding it the manually runned session as positive data and examples of sessions where regression has occurred as negative data. Everytime the automation is then run; the frames or video is used to feed into the CNN for discovering regression.

Machine learning is the newly emerging and pervasive discipline in computer and software science. Everything digitally is shifting its underlying software techniques to AI and ML. Moreover, machine learning has a huge potential in developing good automated tests. A possibility would be to feed two sets of data frames into an ML reinforcement learning algorithm and teach it how to identify when the game is broken and report each such regression. This way, the automated testing will inherently become intelligent, will learn from its own past errors and will develop itself further than any software developer could. Ultimately this is the goal of computer science after all: to fully automate any piece of existing code in a software development project.

Regression can come in many forms and one being in audio. If a game's audio is broken due to a new release of the service that can affect the player's experience just as much as other regression issues. This type of regression detection could be implemented to the existing prototype by for example adding audio recording in addition to input and frame recording. As with comparing frames the audio could be compared in a similar fashion. One possible way of doing audio comparison could be by comparing sound frequencies or by calculating how often each frequency occurs. Here an option of using machine learning for detecting insufficiency between two sets audio could also be beneficial.

6 Conclusion

In the future everything will be stored in the cloud and streamed live. More and more data cannot be located in physical storage anymore. In order to access this stored data fast and error proof, the software must be efficiently and continuously tested. This is already happening for famous streaming platforms such as Netflix and Spotify. However, the gaming industry is still new to this field. Until now, it was common to first download the game on a computing device and then play it. In the recent years games have become extremely complex and they require more processing power in order to be played, power that is missing from many modern computers. To solve this problem, the developers saw the technology advanced enough to make a shift for mobile games to be played on the cloud as well. For this to function properly and for the games to really work correctly, intensive testing must be done throughout the entire lifecycle of the application.

With the numbers of games growing on the cloud gaming service, the testing of their respective software is also increasing. Manual testing is very reliable, but unfortunately it is too strenuous, slow and inefficient. Once a complex game is made compatible with the application it can, however, break later amongst versions of the software. Not only can the game break, but it may also have errors later in the game’s lifecycle. That means that regression can occur in later phases of the game, and that makes it difficult to discover it. In order to solve these issues, there is a need for a new innovative technology and that is automation testing.

The field of automated testing is rich with new practices and theories worthy of scientific investigation. The purpose of this thesis is to solve these newly occurring problems in the mobile gaming industry by presenting automated testing techniques and to explore future possibilities of autonomous playing of the games and image comparison for regression detection.

In this thesis, we have scratched the surface of this new emerging discipline. Utilizing the basic principles of automated testing, this paper analyzes and shows the implementation of a prototype candidate of automated testing, a versatile computational framework suitable for testing regression in mobile games.

In order to demonstrate the functionality incorporated in the prototype, this thesis analyzes its algorithmic particularities and presents the Python version of the script functions. The code records the input pattern of a manual tester playing a working version of a game on the cloud platform. Using these input patterns, the script is able to play the game in the future in the same manner as did the manual tester. During both the manual test run and the autonomous mirrored run, frames are recorded for comparison of these sessions. Using these frames, the autonomous session can be compared to the working run and possible regressions can be detected. If the sessions are identical, no regression has occurred. The results are gathered into a test report which the QA team can then evaluate for further actions.

For that purpose, there are still further improvements to be done, in order for this script to become a fully functioning automated testing scheme. Examples of such additions include:

- Addition of audio processing for regression detection
- Exploitation of machine learning and pattern recognition techniques
- Capturing latency for each input event in order to simulate zero latency during autonomous play
- Possibilities of further improvement of comparing images

The main purpose of the thesis was to show the potential of the prototype as an appropriate automated testing for regression detection in cloud based mobile games. The compactness and the universality make this prototype a strong candidate among automated testing methods. Therefore, this prototype could be suitable to be utilized as part of the mobile gaming industry future additions in the software development and testing process.

References

1 Play Hatch [Internet]. [cited 25 October 2020]; Available from: https://playhatch.com

2 Graham, Dorothy & Fewster, Mark, 2012. Experiences of test automation : case studies of software test automation, Upper Saddle River, NJ: Addison-Wesley.

3 Jose Salvatierra, 2019. Automated Software Testing with Python, Packt Publishing.

4 What is Latency? Stackpath[Internet]. 30 September 2020 [cited 25 October 2020]; Available from: https://blog.stackpath.com/latency/

5 General Python FAQ. Python [Internet]. [cited 25 October 2020]; Available from: https://docs.python.org/3/faq/general.html#id4

6 Android Debug Bridge (adb). Android[Internet]. [cited 25 October 2020]; Available from: https://developer.android.com/studio/command-line/adb

7 What is ADB? How to Install ADB, Common Uses, and Advanced Tutorials. XDA [Internet]. [cited 25 October 2020]; Available from: https://www.xda-developers.com/what-is-adb/

8 Getevent. Android [Internet]. 01 September 2020 [cited 25 October 2020]; Available from: https://source.android.com/devices/input/getevent

9 Linux Input Subsystem userspace API Introduction. Kernel [Internet]. [cited 24 October 2020]; Available from: https://www.kernel.org/doc/html/latest/input/input.html#event-interface

10 Event Codes. Kernel [Internet]. [cited 24 October 2020]; Available from: https://www.kernel.org/doc/Documentation/input/event-codes.txt

11 Multi-touch (MT) Protocol. Kernel [Internet]. [cited 24 October 2020]; Available from: https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt

12 Linux Input drivers v1.0. Kernel [Internet]. [cited 24 October 2020]; Available from: https://www.kernel.org/doc/Documentation/input/input.txt

13 Sendevent.c. Git [Internet]. [cited 24 October 2020]; Available from: https://android.googlesource.com/platform/system/core/+/android-5.0.2_r3/toolbox/sendevent.c

14 Subprocess management. Python [Internet]. [cited 24 October 2020]; Available from: https://docs.python.org/3/library/subprocess.html

15 About. OpenCV [Internet]. [cited 24 October 2020]; Available from: https://opencv.org/about/

16 diffimg. Github [Internet]. 30 March 2016 [cited 24 October 2020]; Available from: https://github.com/nicolashahn/diffimg

17 Aapt. Pypi [Internet]. [cited 24 October 2020]; Available from: https://pypi.org/project/aapt/

18 Convolutional Neural Network (CNNs/ConvNets). Github [Internet]. 21 October 2020 [cited 24 October 2020]; Available from: https://cs231n.github.io/convolutional-networks/

19 Pedro Coelho, L., Brucher, M. & Richert, W., 2018. Building Machine Learning Systems with Python - Third Edition, Packt Publishing.

20 Alencar, Paulo, 2012. Handbook of research on mobile software engineering : design, implementation, and emergent applications, Hershey, PA: Engineering Science Reference.

21 Turnquist, Greg Lee, 2011. Python testing cookbook : over 70 simple but incredibly effective recipes for taking control of automated testing using powerful Python testing tools, Birmingham, U.K: Packt Pub.

22 Kevin Bowersox, 2017. Continuous Integration (CI) With Jenkins - Quality Assurance and Automated Testing, Infinite Skills.

23 Crispin, Lisa & Gregory, Janet, 2009. Agile testing : a practical guide for testers and agile teams, Upper Saddle River, N.J: Addison-Wesley

24 Dustin, Elfriede, Garrett, Thom & Gauf, Bernie, 2009. Implementing automated software testing : how to save time and lower costs while raising quality, Upper Saddle River, N.J: Addison-Wesley.

25 Anon, 2005. Game testing all in one, Boston, Mass: Thomson/Course Technology

26 Google Play Store [Internet]. [cited 25 October 2020]; Available from: https://play.google.com/store/apps/details?id=live.hatch

27 Play Hatch Press Kit [Internet]. [cited 25 October 2020]; Available from: https://hatch.frontify.com/d/ib1gZE69euP4/hatch-press-kit#/hatch-press-kit/press-images