Tampereen ammattikorkeakoulu

Kuvantunnistukseen pohjautuva automaatiotestaus Unity-ympäristössä

Antti Tuomisto

OPINNÄYTETYÖ
Joulukuu 2019

Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

TUOMISTO, ANTTI:
Kuvantunnistukseen pohjautuva automaatiotestaus Unity-ympäristössä

Opinnäytetyö 31 sivua
Joulukuu 2019

Tämän opinnäytetyön tavoitteena on selvittää miten Unity-pelimoottorilla tehtyjen mobiilisovellusten testauksen eri osa-alueita voitaisiin automatisoida. Kuvantunnistuksella pyritään vastaamaan Unityn asettamiin ongelmiin perinteisessä mobiilielementtien tunnistuksessa. Opinnäytetyössä parannettiin Traplight Oy -nimisen yrityksen automaatiotestausjärjestelmän kehitysympäristöä, sekä selvitettiin järjestelmässä vaadittavia työkaluja ja niiden toimintaa.

Opinnäytetyössä viitataan joihinkin yritykselle suunnitellun automaatiotestauksen järjestelmän kohtiin. Tämä ei kuitenkaan ole sellaisenaan yrityksen käytössä, vaan sitä kehitetään edelleen. Opinnäytetyö on mahdollistanut jatkokehityksen antamalla projektille suuntaa ja työn avulla on onnistuttu paikantamaan, sekä rajaamaan nykyisistä kehitystyökaluista toimivimmat.

Suurimmat haasteet työssä liittyivät kuvantunnistuksen menetelmiin ja niiden tuottamien tulosten varmuuteen. Myös yrityksen oma Unityyn integroitu järjestelmä aiheutti ylimääräisiä haasteita kehitystyöhön. Työ kuitenkin todisti, että tämänkaltainen järjestelmä voidaan kehittää ainakin perinteisiin Unity-projekteihin, mutta myös osaksi erillistä pelinkehitysjärjestelmää.

Asiasanat: testaus, automaatiotestaus, unity, kuvantunnistus

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme of Business Information Systems
Option of Game Production

TUOMISTO, ANTTI:
Image recognition based automated testing in Unity environment

Bachelor's thesis 31 pages
December 2019

The objective of this thesis was to find out about different possibilities related to automated testing of mobile applications developed in Unity. Image recognition was used to answer problems created by Unity that are not apparent in regular mobile test automation. The purpose of this thesis was to improve the automation development practices for Traplight Oy and to explore different tools required to build a unique testing system.

This thesis contains references to different parts of the test automation system that was created for this organization. This system is not fully in use and is still under development. The work done for this thesis has however allowed further development and steered the project into right direction by providing the development tools that work best.

The greatest challenges in the development have come from the different methods of image recognition. Correct results were hard to achieve and most of the findings still created a lot of false positives. Also, the integrated company development platform created some additional challenges. Despite the problems the work done has proven that this kind of system can be developed, and thesis will also explore some alternative ways to tackle these issues.

Key words: testing, test automation, unity, image recognition

SISÄLLYS

LYHENTEET JA TERMIT

Android Manifest	Android Manifestilla tarkoitetaan AndroidManifest.xml tiedostoa, joka tarjoaa laitteen sisäisille ominaisuuksille tietoa laitevaatimuksista, sovelluksen vaatimista käyttöluvista ja komponenteista.
JSON	Lyhenne sanoista JavaScript Object Notation on yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen.
ORB	Patentoimaton algoritmi kuvan ominaisuuksien tunnistamiseen. ORB on lyhenne sanoista oriented FAST and rotated BRIEF.
Paketti	Pakettinimi vastaa yleensä projektin nimiavaruutta ja muodostaa tunnisteen, jota käytetään määrittelemään sovelluksen koodin sijainti projektia käännettäessä.
Python	Opinnäytetyössä hyödynnetty ohjelmointikieli.
SIFT	Kuvan mitoista riippumaton algoritmi ominaisuuksien tunnistamiseen. SIFT on lyhenne sanoista scale-invariant feature transform.
SURF	Nopeampi vaihtoehto SIFT algoritmille. SURF on lyhenne sanoista speeded-up robust features.

1 JOHDANTO

Testaus on tärkeä osa ohjelmistokehitystä ja automaatiotestaus on testauksen uudempia keskeisiä suuntauksia. Tämä opinnäytetyö pohjautuu pyrkimykseen kehittää testausprosesseja ja automaatiotestauksessa hyödynnettäviä työkaluja tamperelaiselle pelialan yritykselle nimeltä Traplight Oy. Työn tarkoitus on vähentää tarvetta manuaaliselle testaukselle, luomalla Unity-pelimoottorilla kehitetyille mobiilipeleille soveltuva automaatioon perustuva testausjärjestelmä. Järjestelmän pohjalta tullaan tarvittaessa luomaan ratkaisuja osaksi toimeksiantajan tuotantoprosessia.

Opinnäytetyössä tutustutaan myös projektiorganisaatioiden rakenteeseen, sekä testauksen yleisimpiin menetelmiin. Opinnäytetyön perimmäinen tarkoitus on tutustua testauksen tarpeisiin ja tarkastella niiden pohjalta vaihtoehtoisia menetelmiä automaatiotestauksen käyttöönottolle mobiilialustalähtöisessä Unity-pelinkehitysympäristössä. Keskeinen osa työn tarpeellisuuden määrittelyä liittyykin Unityn luomiin rajoitteisiin ajettaessa automaatiotestejä ja tunnistettaessa graafisen käyttöliittymän eri osia. Näihin rajoituksiin pyritään vastaamaan kuvantunnistuksen keinoin.

Lopuksi otetaan selvää mihin tämän kaltainen järjestelmä voi parhaimmillaan kehittyä. Lisäksi käydään läpi työn aikana ilmenneet löydökset ja perehdytään pikaisesti muihin mahdollisiin automaatiotestaukseen liittyviin vaihtoehtoisiin käyttötapoihin ja palveluihin.

2 PELINTESTAUS

2.1 Yleisesti

”Ohjelmistotestaus tarkoittaa työtä, joka tehdään sen varmistamiseksi, että toteutettavasta ohjelmistotuotteesta tulee toivotun kaltainen ja että kaikki siihen valmiiksi saadut ominaisuudet varmasti toimivat niinkuin on tarkoitus” (Kasurinen, 2013). Testaus lähteekin siitä ajatuksesta, että kehitettävä tuote ei ole koskaan täydellinen tai valmis. Tämän takia kaikkia sovelluksessa ilmenneitä virheitä ei tarvitse välttämättä edes korjata, mutta testaajaa voidaan vaatia projektin viitekehyksen tiimoilta analysoimaan ja priorisoimaan tehtävistä kaikkein kriittisimmät (Charles P. Schultz, 2017).

Testauksen toteutustavat ovat yksilöllisiä ja vaihtelevat hyvin paljon sitä toteuttavien tahojen välillä, minkä takia yhtä selkeää työtehtävää on vaikea määritellä. Testaustyön suorittaminen on erityisesti tärkeää projektin aikana. Julkaisun jälkeisen testauksen on arvioitu maksavan yrityksille yli 90%, siitä mitä testauksesta syntyvä hinta olisi toteutettuna kehitysprosessin aikana (Kasurinen, 2013). Testaus tulee projektin aikana suunnata erityisesti monimutkaisiin osa-alueisiin, sekä sellaisiin moduuleihin, joiden ennestään tiedostetaan olevan alttiita hajoamaan. Työprosessi vaatii siis ennakointia ja analyysien valmistelua (Kasurinen, 2013). Testaus ei ole vain järjestelmän visuaalisuuden ja koodin toimivuuden tarkastusta, vaan siihen kuuluu myös käytettävyyden analysointi, käyttäjätestien laatiminen ja palautteen antaminen kehitystiimille. Laadunvalvonnan vastuulla on loppukädessä koko projektin ulkoasu ja julkaistavan tuoteen menestymisen mahdollistaminen. Testaajalta vaaditaankin jatkuvaa projektin hallintaa, sen ymmärtämistä ja niiden pohjalta tehtävien päätösten laatimista.

2.2 Staattinen ja dynaaminen testaus

Testauksesta voidaan eriyttää menetelmiä sen mukaan, millä keinoin ja missä projektin vaiheessa niitä toteutetaan. Näitä ovat esimerkiksi staattinen ja

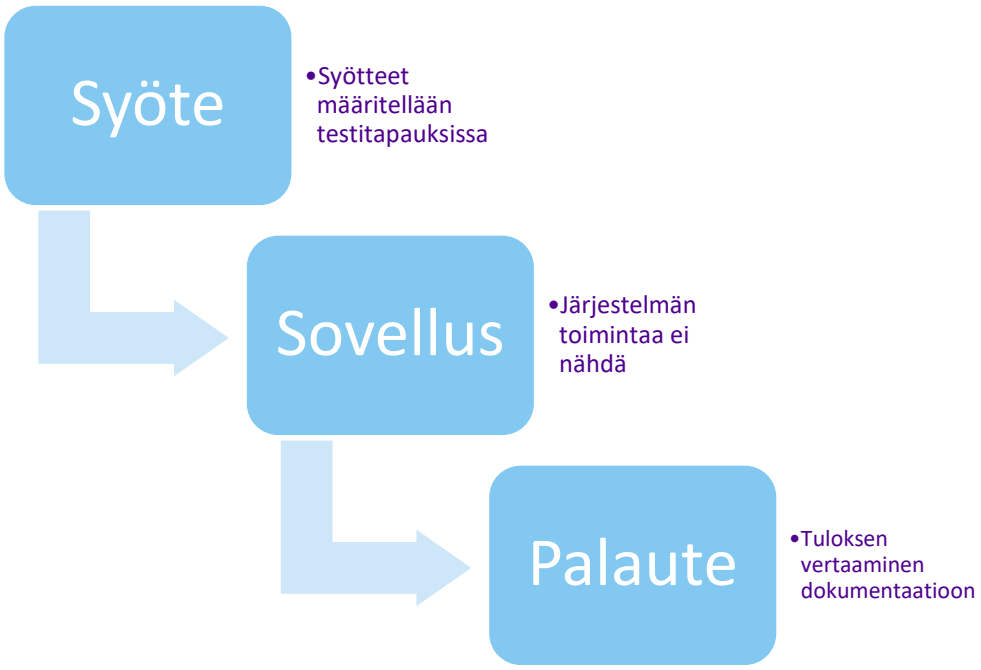
dynaaminen testaus. Staattisiin menetelmiin kuuluvat järjestelmän analysoiminen sekä ongelmien havainnointi yleisellä tasolla ja se voidaan aloittaa arkkitehtuurisuunnitelman pohjalta (Kasurinen, 2013). Staattinen testaus pyrkii poistamaan kaikkein alttiimmat virhetilat jo ennen kuin testausta on virallisesti aloitettu.

Dynaamisen testauksen piiriin voidaan laskea perinteisemmät testaustavat, kuten yksikkötestaus, integrointitestaus ja kuormitustestaus, joiden tarkoitus on tarkastella järjestelmän toimivuutta kaikissa mahdollisissa tilanteissa. Staattinen testaus tulee yleensä organisaatioille halvemmaksi, koska se tehdään aikaisemmassa vaiheessa ja sillä vähennetään ennakoivasti ongelmia, jotka syntyisivät myöhemmin dynaamisen testauksen aikana.

2.3 Mustalaatikkotestaus

Lähes kaikki pelintestaus on sovelluksen ulkopuolelta tapahtuvaa mustalaatikkotestusta. Sen toteutustapa on hyvin yksinkertainen ja perustuu siihen, että testihenkilö ilman minkäänlaista informaatiota lähdekoodista varmistaa että peli tai sovellus toimii (Charles P. Schultz, 2017). Ajankäytön ja resurssienhallinnan osalta tämän kaltainen testaus ei ole kovin tehokas keino monimutkaisempia ohjelmistojärjestelmiä testatessa.

Tavallisessa organisaatiossa testaaja suorittaa musta laatikko -testin ohjelmoijan toteuttamalle ominaisuudelle (kuva 1). Testi aloitetaan syöttämällä ihanteellisimmat komennot testattavalle alustalle ja arvioimalla sen toimivuutta (Kasurinen, 2013). Tämän jälkeen siirrytään monimutkaisempiin käskyihin ja pyritään kattamaan kaikki ääritapaukset. Mikäli annetut syötteet johtavat virhetilaan, tehdään asiasta lopuksi ilmoitus lomakkeen muodossa.



Kuva 1. Musta laatikko testausprosessi vaiheittain

Musta laatikko -testauksen menetelmiä voidaan yleensä hyötykäyttää kaikissa projektin vaiheessa, kunhan järjestelmän kanssa voidaan kommunikoida. Musta laatikko -testauksen tukena on hyvä olla dokumentaatiota siitä, miten järjestelmän tulisi käyttäytyä missäkin testattavassa tilantessa. Testaustapa saa nimensä siitä, että testien aikana ei yleensä nähdä sitä, mitä järjestelmän sisällä tapahtuu (Kasurinen, 2013).

2.4 Valkoinen laatikko -testit

Valkoinen laatikko -testauksen tukena testaajalla on käytössään testattavan sovelluksen lähdekoodi. Tämänkaltainen testausmenetelmä sopiikin paremmin esimerkiksi projektin ohjelmoijalle testattaessa oman koodin toimivuutta. Valkoinen laatikko -testauksen käyttö mahdollistaa pelin toimintojen kiihdytetyn testauksen ja sillä voidaan parhaiten varmistua koodin uudelleenkäytettävien moduulien toimivuudesta (Charles P. Schultz, 2017). Tämänkaltaisen testauksen haittana on kuitenkin se, että testausympäristö on aina ideaalissa tilassa ja siihen harvoin vaikuttavat ulkoiset tekijät, kuten esimerkiksi testauslaitteiston heikko suorituskyky tai rajallinen muistin käyttö.

3 TESTAUS OSANA PROJEKTIKEHITYSTÄ

3.1 Testaajan rooli

Testauksessa tärkeää on pystyä toistamaan samaa toimintoa useita kertoja, mikä vaatii testaajalta pitkäjänteisyyttä. Testaajat ovat loppukädessä vastuussa lopullisesta julkaistavasta tuotteesta ja sen toiminnasta. Virheet tulee huomata hyvissä ajoin, sillä tuotteen laadun laiminlyöminen voi johtaa suurempiin ongelmiin myöhemmissä kehitysprosessin osissa. Ohjelmoijien on myös helpompi korjata ongelmat, kun ominaisuus on vielä tuoreena muistissa. Suuremman organisaation osana testaajien olisi hyvä pitää yllä tehokasta kommunikaatiota muun kehitystiimin kanssa ja olla myös perillä siitä, mistä ominaisuudesta kukin kehitystiimin jäsen on vastuussa. Tuotantotiimille kommunikoinnissa käytetään usein projektihallinnan työkaluja. Testaajan on työkalujen avulla helpompi raportoida ongelmat kehitystiimille. Raportointiin kuuluu ongelman vakavuuden arviointi ja epäonnistuneeseen testiin johtavan toimintaketjun varmentaminen. Vakavuuden arviointi on erityisen tärkeää, jotta muut kehitystyöhön osallistuvat tahot voivat priorisoida sen avulla omaa työtaakkaansa.

Testausta on projektin kannalta tärkeää pitää yllä koko kehitysprosessin ajan. Jos testauksen jättää projektin loppuvaiheille on tuolloin hankalampaa korjata suurempia, koko projektin kattavia ongelmia, jotka voivat olla hyvinkin kriittisiä lopullisen onnistumisen kannalta. Vaikka pelintestausta on harjoitettava noudattamalla organisaatiokohtaisia käytäntöjä esimerkiksi raportoinnissa, on testaajilla yleensä luonteeseen pohjautuvia yksilökohtaisia eroavaisuuksia (Charles P. Schultz, 2017).

3.2 MBTI

Gametesting All in One -teoksessa erotellaan testaajat karkeasti kahteen eri ryhmään, Myers-Briggs Type Indicatorin avulla (MBTI). Tämä teoria jakaa ihmiset kahteen psykologiseen suuntaukseen, joita ovat tuomitsija, englanniksi judger ja

tarkkailija eli perceiver (Charles P. Schultz, 2017). Tuomitsija-tyyppinen ihminen vaatii teoksen mukaan parhaiten toimiakseen hyvin jäsennellyn, järjestelmällisen ja ennalta-arvattavan työympäristön. Hän suorittaa testit ennalta määritellyn kaavan mukaan ja kiinnittää paremmin huomiota testattavan tuotteen mahdollisiin muutoksiin ja erinäisiin virheisiin. Tämänkaltaisten testaajien on myös helpompi huomata mahdolliseen virheeseen johtanut tapahtumaketju.

Tarkkailija taas keskittyy kokemaansa ja mieluummin näkee asioiden ratkoutuvan omia aikojaan. Tämänkaltainen kaoottisempi testaustapa poistaa testien suorituksesta ennalta-arvattavuutta ja kehittää testaajan luovaa ajattelua, sekä ongelmanratkaisukykyä.

4 AUTOMAATIOTESTAUS

4.1 Automaation tarpeellisuus

Automaatiotestauksella tarkoitetaan toistuvien testitapausten varalle erikseen kehitettyjä automaattisia testauksen työkaluja. Automaatiotesteillä voidaan vastata esimerkiksi mobiilialustan laajan laitepohjan luomiin haasteisiin, sillä testit mahdollistavat saman sovelluksen vaivattoman testaamisen useilla alustoilla. Kasurinen esittää kirjassaan että keskimäärin 10% kaikesta testaustyöstä on automatisoitu ja vaikka kirjan kirjoittamisen jälkeen ala on kehittynyt eteenpäin, on erikseen nimetyn testaushenkilöstön työpanos projektiorganisaatiossa edelleen tarpeellista sovelluskehityksen kannalta (Kasurinen, 2013). Automatisoitu testaus on ihmistä tehokkaampaa, mutta se vaatii paljon aikaa testien valmisteluun. Automaatiotestauksen on myös mahdotonta ottaa abstraktia lähestymistapaa sovellusten testaukseen. Parhaimmillaan testausautomaatio täydentää manuaalista testausta ja vapauttaa testaajan muihin tehtäviin (Kasurinen, 2013).

4.2 Automaatiotestin valmistelu

Testin valmistelu aloitetaan määrittelemällä tarkasti mihin suoritettavalla tehtävällä pyritään. Testi tulee suunnitella siten, että se ei vaadi suorituksen aikana minkäänlaista väliintuloa, vaan pystyy suoriutumaan tehtävästä itsenäisesti. Mikäli automaatiotestejä ajettaessa tarvitsee määritellä ajon aikana hyödynnettäviä parametrejä, tulisi niiden olla mahdollisimman yksiselitteisiä työn helpottamiseksi.

Kun ohjelmassa ilmennyt vika on saatu korjattua, tulee saman ominaisuuden toimivuus kyetä takaamaan myös tulevaisuudessa. Testaaja voi itse tarvittaessa kirjoittaa mekaaniseen toistoon automaatiotestin, joka ajetaan itsenäisesti tietyin aikavälein. Samaa testiä voidaan varmuuden vuoksi ajaa vaikka koko loppu tuotteen kehityskaaren ajan. Testin tarkoitus voi olla esimerkiksi tietyn graafisen käyttöliittymän elementin tunnistaminen ja sen oikeanlaisen toimivuuden

varmistaminen. Graafisen käyttöliittymän automaatiotestit voidaan kuitenkin tehdä vasta, kun testattavan ominaisuuden toiminnallisuus on ensin manuaalisesti varmennettu.

Mobiilipelit asettavat sovelluksen testaukseen tilakohtaisia haasteita, joihin vastaaminen vaatii testaajilta hieman esityötä. Tiettyihin testattaviin tiloihin pääsemiseksi vaaditaan käyttäjiltä välillä pitkiäkin pelisessioita. Tähän ratkaisu voi olla esimerkiksi sellaisen komentosarjan kirjoittaminen, joka pelaa peliä automaattisesti tarvittavaan vaiheeseen asti. Tämä voi olla aikaa vievää, jos pelin läpi pelaaminen kestää esimerkiksi useita viikkoja. Toinen tapa ongelman ratkaisemiseksi on asettaa laitteisiin testattava ympäristö manuaalisesti valmiiksi, esimerkiksi ylläpitäjän oikeuksia hyödyntämällä. Tämä pakottaa testaajan aktiivisesti tarkkailemaan suorituksen kulkua, hidastaa automaattista testausprosessia ja luo haasteita nopeisiin etäyhteydellä ajettuihin testeihin. Kolmas vaihtoehto on testata erillistä versiota sovelluksesta, joka on määritelty alkamaan vaaditusta tilasta. Tällöin testi taas voi poiketa liikaa aidosta käyttötilanteesta.

4.3 Tulokset

Kun testi on suoritettu, on testiautomaatiojärjestelmän hyvä tarjota ajetusta testistä mahdollisimman tarkka raportti. Tulokset tallennetaan myöhempää tarkastelua varten ja niiden pitää sisältää oleellista tietoa ajetusta testistä. Tuloksiin kerätään tuloste ilmenneistä virheitä ja tarvittaessa voidaan ottaa esimerkiksi kuvakaappaus tilasta, johon testi kaatui, tai nauhoittaa koko testi videolle. Testaustapojen muuttuvan luonteen ja sovelluksen monimutkaistumisen takia myös automaatiotestejä täytyy aktiivisesti ylläpitää ja päivittää.

Suurin osa testausautomaation tekemisestä määräytyy sen mukaan, mitä työkalua sen toteuttamiseen käytetään (Kasurinen, 2013). Testaustyökaluja voidaan parhaiten käyttää regressiotesteihin, eli kun halutaan varmistaa että mikään aikaisemmin kehitetty sovelluksen osa-alue ei ole hajonnut viimeisimpien muutosten jälkeen.

5 AUTOMAATIOJÄRJESTELMÄN TYÖKALUT

5.1 PyTest

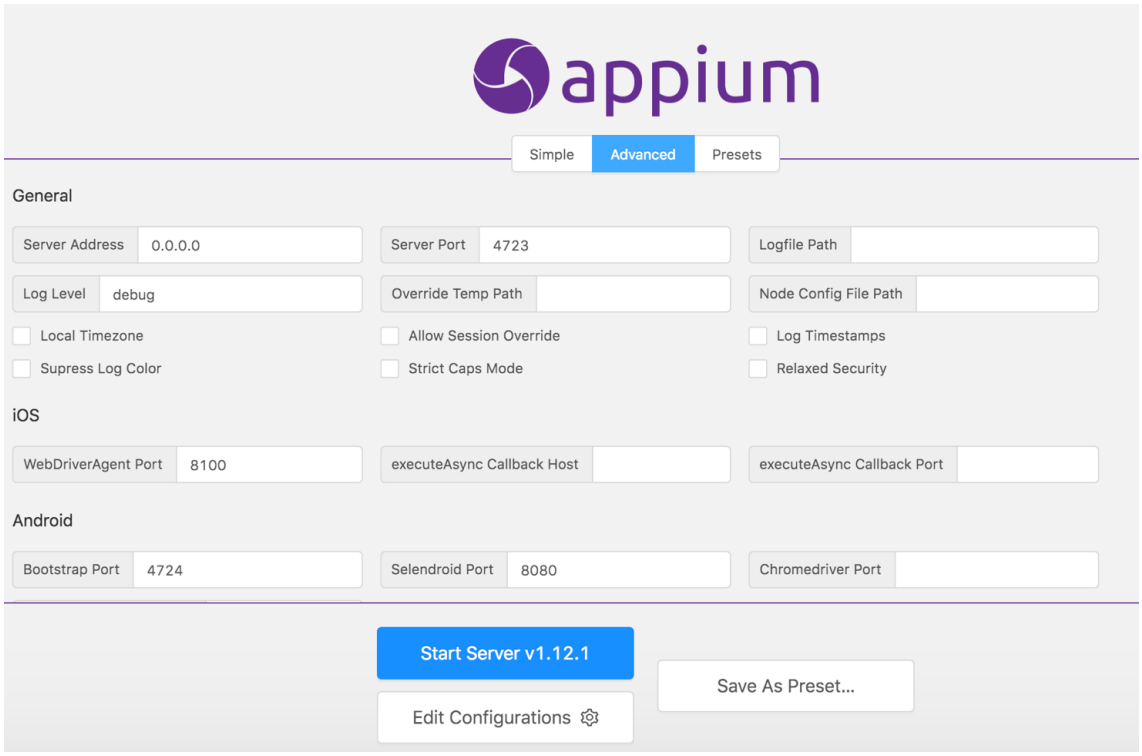
PyTest on yksinkertainen ja helposti laajennettava Python-ohjelmointikieltä hyödyntävä avoimen lähdekoodin testityökalu (PyTest, Full pytest documentation, 2004-2017). PyTest on valittu opinnäytetyötä varten kehitetyn automaatitestausjärjestelmän alustaksi, koska sitä on helppo käyttää ja se skaalautuu vaivatta kuvantunnistuksen tarpeisiin. PyTest tunnetaan yleisimmin alustana yksikkötesteille, mutta sillä voidaan ulkopuolisten järjestelmien ja Python ohjelmointikielen modulaarisuuden avulla ajaa vaivatta myös hyväksymistestejä. Järjestelmä sisältää muutamia määreitä, jotka komentosarjojen tulee järjestelmän toimimiseksi täyttää. Testit suoritetaan Python -kielelle ominaisessa järjestyksessä. Kaikki testit voidaan tarvittaessa ajaa samasta luokasta, jolloin järjestelmä hakee ensimmäisenä ajettavasta koodista kaikki funktiot, jotka alkavat prefiksillä ”test_” ja suorittaa ne. Virheilmoituksia ja tulosteita ei oletuksena näytetä ajon aikana, vaan ne napataan talteen ja näytetään testien jälkeisessä tulosteessa (PyTest, Capturing of the stdout/stderr output). Kun testi on suoritettu, kertoo tuloste kuitenkin ensimmäisenä menivätkö testit asetetuilla parametreillä läpi vai ei.

5.2 Appium

Appium on avoimen lähdekoodin työkalu testausautomaatioon. Appiumin avulla voi automatisoida muun muassa web- ja hybridiratkaisuja. (Appium, Introduction to Appium, 2019). Tämän opinnäytetyön esimerkeissä keskitytään kuitenkin Unity pelimoottorilla kehitettyihin mobiilisovelluksiin. Appiumin toiminta on järjestelmäriippumatonta, jolloin sen ohjelmointirajapintaa hyödyntävät testiskriptit toimivat sekä iOS-, Android-, että Windows-testitapauksissa. Työmäärän kannalta tämä on hyödyllistä, sillä testejä ei tarvitse kirjoittaa joka alustalle erikseen. Appium tukee useita ohjelmointikieliä, kuten Java, Python, C#, Ruby, JavaScript ja PHP. Appium mahdollistaa myös vaivattomasti emulaattoreiden käytön testauslaitteena.

5.2.1 Appiumin käyttöönotto

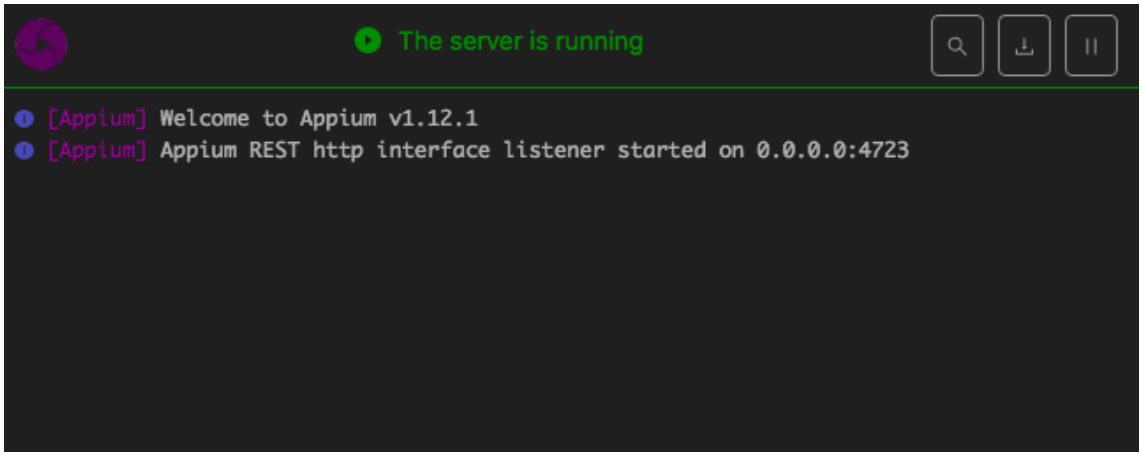
Yksinkertaisuudessaan käyttäjä antaa Appium-rajapinnan kautta palvelimelle komentoja ja Appium suorittaa sille annetut komennot määriteltyä ajuria käyttämällä. Esimerkiksi Android- ja iOS-alustoja hyödynnettäessä tulee Appium määritellä käyttämään eri ajureita. Android-sovellukset käyttävät UiAutomator 2 -ajuria ja iOS- sekä tvOS-sovellukset taas XCUITest-ajuria. Käyttöönotto aloitetaan käynnistämällä sovellus ja määrittelemällä Appiumin käyttämät porttinumerot (kuva 2). Oletuksena portti on 2743. Mikäli halutaan ajaa useita testejä rinnakkain, voidaan sitä varten käynnistää useita palvelimia, jolloin uusi porttinumero tulee määritellä uusiin Appium-instansseihin. iOS-laitteita varten on myös hyvä määritellä WebDriverAgent-portti ja Android-laitteita varten Bootstrap-portti.



Kuva 2. Appium sovellus v.1.12.1

Appium voidaan käynnistää tarvittaessa myös komentoriviltä komennolla ”appium”. Appium-palvelin aukeaa oletusportilla 4745. Samaan komentoon on hyödyllistä määritellä esimerkiksi toinen porttinumero lisäämällä kutsun perään komentorivilippu ”-port <portin numero>”. Kaikki palvelinparametrit on listattu

virallisessa Appium-dokumentaatiossa. Onnistuneen käynnistyksen päätteeksi Appium kertoo, mikä versio sovelluksesta on käytössä ja mitä porttia se kuuntelee (kuva 3).



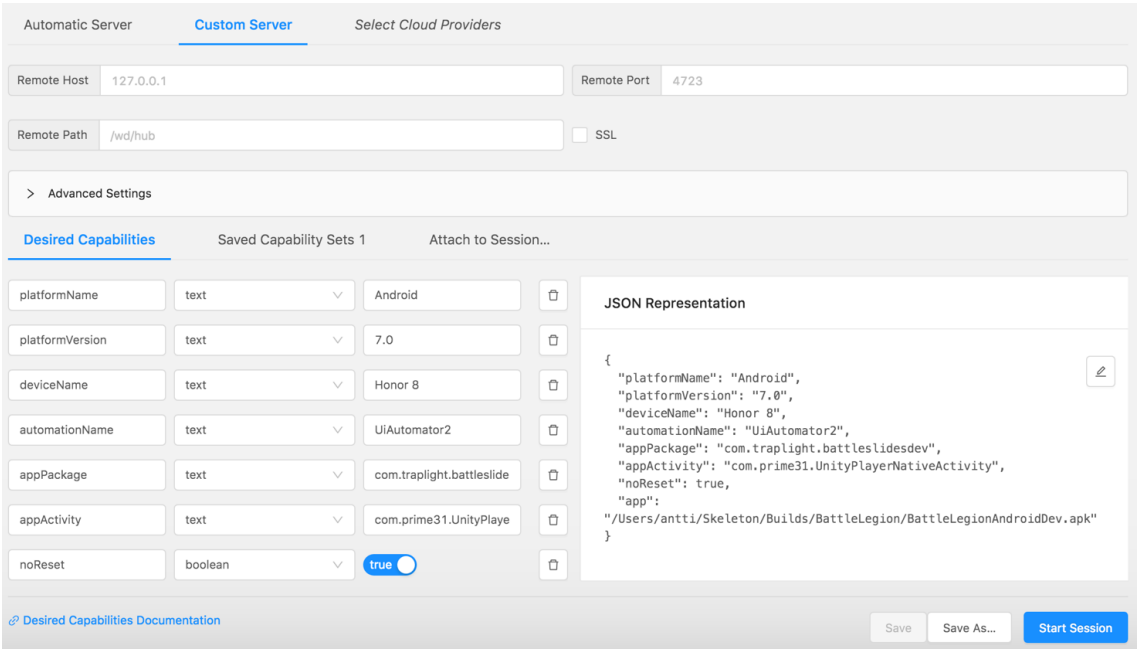
Kuva 3. Appiumin käynnistys sovelluksessa

Pohjimmiltaan Appium on perinteinen web-palvelin joka toimii REST API -periaatteella. Asiakaspääte ottaa yhteyden Appium-palvelimeen. Palvelin kuuntelee sille annettuja komentoja ja suorittaa ne sille määritellyssä laitteessa (Appium, Introduction to Appium, 2019). Testaustarpeitten mukaan voi olla järkevää erottaa Appiumin palvelin erilliselle koneelle. Tämä mahdollistaa sen, että testejä voi etäohjata eriyttämällä testilaitteet saman laitteen komentoon. Tämä parhaimmillaan vapauttaa koneen suorituskykyä, josta on hyötyä esimerkiksi emulaattoreita ajettaessa. Palvelimen fyysinen eriyttäminen auttaa myös selkeyttämään työprosessia kokonaisuutena jakamalla sitä samalla myös fyysisiin kokonaisuuksiin.

Jos halutaan ajaa useita samanaikaisia testejä eri laitteilla, voidaan luoda useita Appium-palvelimia eri porttinumeroilla, sekä parametreillä. Jotta yhteys Appium-palvelimeen voidaan muodostaa, tulee testiskriptissä ensin määritellä Appiumin saataville pakollisia laite- ja ohjelmistotietoja.

5.2.2 Sovelluksen ominaisuuksien määrittely

Appium-työpöytäsovellus tarjoaa helpot tekstikentät erinäisten parametrien määrittelyyn (kuva 4). Automaatiot suoritetaan Appiumin kanssa istuntokohtaisesti. Tavallisesti ohjelmakoodi pyrkii aloittamaan istunnon lähettämällä Appium palvelimelle post/session-pyynnön mukana olion, joka sisältää laitekohtaiset ominaisuudet. Palvelin todentaa olion sisällön ja vastaa istunnon tunnisteella. Tämä toimii istunnon aloituksena, jonka jälkeen on mahdollista lähettää lisää komentoja laitteelle. Unityn tuottamat ongelmat elementtien tunnistuksessa voidaan todentaa nopeimmin muodostamalla testiyhteys testattavaan laitteeseen Appiumin-työpöytäsovelluksessa.

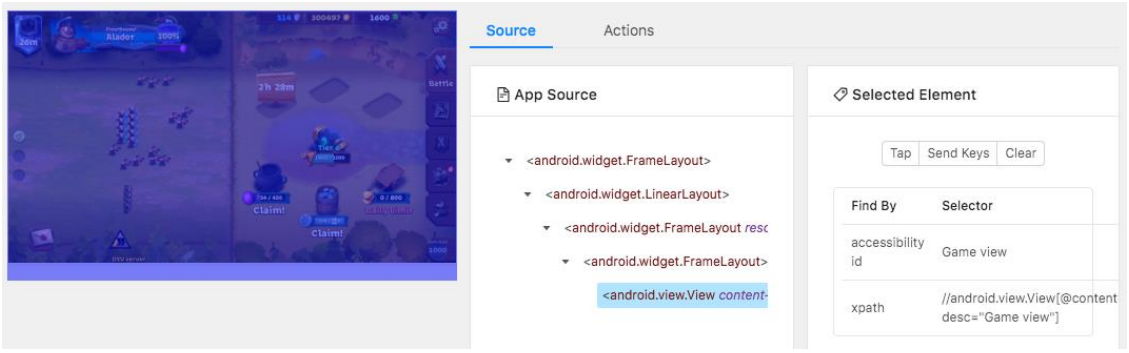


Kuva 4. Avainsana ja arvo-parien asettaminen

Desired capabilities on avainsana ja arvo -pari, joka on koodattu JSON-olioon. Se kertoo Appium-palvelimelle, mitä laitetta halutaan käyttää ja millä konfiguraatioilla. Elementtien ongelman visualisoimiseksi ei tarvitse kuitenkaan kirjoittaa testiskriptiä, vaan voidaan avata istunnon aloitusikkuna painamalla suurennuslasipainiketta. Tästä aukeaa ikkuna, jossa voidaan syöttää muuttujat niille määriteltyihin kenttiin. Näin saadaan helposti avattua sovellus halutussa laitteessa.

Yleisimmät konfiguroitavat muuttujat ovat nimeltään platformName, platformVersion, deviceName, automationName, appPackage, appActivity, noReset ja app (Appium, Introduction to Appium, 2019). Ensimmäiset kolme määrittävät laitekohtaiset vaatimukset. Appium ei käytä deviceName-muuttujaa laitteen tunnistamiseen, mutta se tulee appium dokumentaation mukaan täyttää. platformName määrittää laitteen käyttöjärjestelmän ja platformVersion sen versionumeron. Esimerkissä käytetty versio on Android Nougat 7.0. Automaatiossa käytettävät ajurit määritellään automationName-muuttujalla. Esimerkissä käytetään UIAutomator2-ajuria. Appiumissa voi käyttää myös Selenoid-ajuria tai vanhempaa versiota Ulautomatorista, joita ei virallisesti tueta sovelluksen uudemmissa versioissa. Mikäli sovellus halutaan asentaa laitteelle suorituksen yhteydessä on hyvä määritellä app muuttuja, joka sisältää polun testattavaan .apk-tiedostoon. Appiumin dokumentaatio myös suosittelee näiden lisäksi käytettävän Android activity -muuttujaa sekä paketin nimeä. Jos näitä ei täytetä, yrittää Appium määritellä ne automaattisesti sovelluksen manifestista. Jos taas suoritettavien testien aikana tulee odottaa pitkiä aikoja tilassa, jossa client ei anna komentoja, on hyvä määritellä newCommandTimeout-muuttuja. Tämän avulla voidaan millisekunneissa määrittää, kuinka kauan tulee odottaa ennen kuin sovelluksen todetaan pysähtyneen ja testin suoritus lopetetaan. Activity kuvastaa käyttäjälle esitettävää tasoa Androidille asennetusta sovelluksesta. Se voidaan ymmärtää Androidin sisässä sovelluksen käytössä olevana ikkunana. Android voi vaihtaa ikkunoita sovellusten suorituksen aikana. Kun halutut ominaisuudet on määritelty, voidaan aloittaa istunto Start Session -painikkeella.

Appium etsii tai asentaa sovelluksen ja avaa sen laitteessa. Tämän jälkeen aukeaa ikkuna, johon voidaan päivittää sovelluksen näkymästä ruudunkaappaus (kuva 5). Ruudunkaappaus näkyy ikkunan vasemmalla puolella ja jokainen elementti voidaan erikseen korostaa sinisellä taustalla. Laajennettaessa pudotusvalikkoa huomataan Unity-kehitysympäristössä luotujen pelien ongelmallisuus käsiteltäessä elementtejä.



Kuva 5. Kuvankaappaus Appium-sovelluksessa

Elementti täyttää koko ruudun tilan, jolloin sovelluksen sisältämiä yksittäisiä graafisen käyttöliittymän painikkeita ei voida hyödyntää testiskriptejä kirjoitettaessa.

6 AUTOMAATTISEN TESTAUSJÄRJESTELMÄN TOTEUTUS

6.1 Kuvan lukeminen

Unityn asettamia rajoitteita voidaan paikata esimerkiksi kuvantunnistuksen keinoin. Opinnäytetyön ohessa toteutettu järjestelmä hyödyntääkin OpenCV-ohjelmointikirjastoa (Open Source Computer Vision Library) käyttöliittymän komponenttien tunnistuksessa. OpenCV sisältää useita satoja konenäön algoritmeja (OpenCV, OpenCV documentation index). Kirjaston moduuleista hyödynnetään tässä työssä kuitenkin pääasiassa niitä, jotka liittyvät kuvankäsittelyyn, kuvantunnistukseen ja niiden vertailuun, sekä 2D-ominaisuuksien viitekehystä.

Kuvantunnistusta hyödynnettäessä elementtien löytämiseksi, täytyy käyttäjällä olla kuvakirjasto, joka sisältää kuvan haettavasta objektista, sekä ruudunkaappaus, johon kuvaa verrataan. Jotta kuvaa voidaan hyödyntää openCV-kirjaston avulla, tulee se ensin lukea levyltä ja tallentaa omaan muuttujaansa.

```
template_rgb = cv.imread('button.png', 0)
```

Imread ottaa tässä tapauksessa parametrin 0, joka muuttaa luetun kuvan mustavalkoiseksi tunnistuksen nopeuttamiseksi. Värikanavien vertaaminen ei ole välttämätöntä, mutta saattaa tuottaa tarkempia tuloksia.

```
Screenshot = self.driver.get_screenshot_as_png()
img = self.convert_to_numpy(screenshot)
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

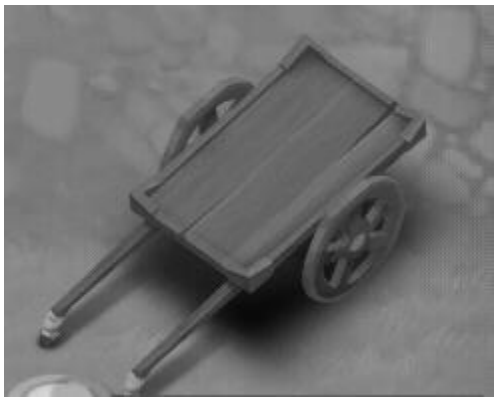
Haettavan kuvan lisäksi tarvitaan varsinainen mallikuva, johon sitä verrataan. Esimerkissä haetaan ruudunkaappaus Appiumin avulla. Kuva muutetaan käsiteltävään muotoon ja kuten edellisessä esimerkissä siitä tehdään harmaansävyinen.

6.2 Kuvien ominaisuuksien vertaaminen

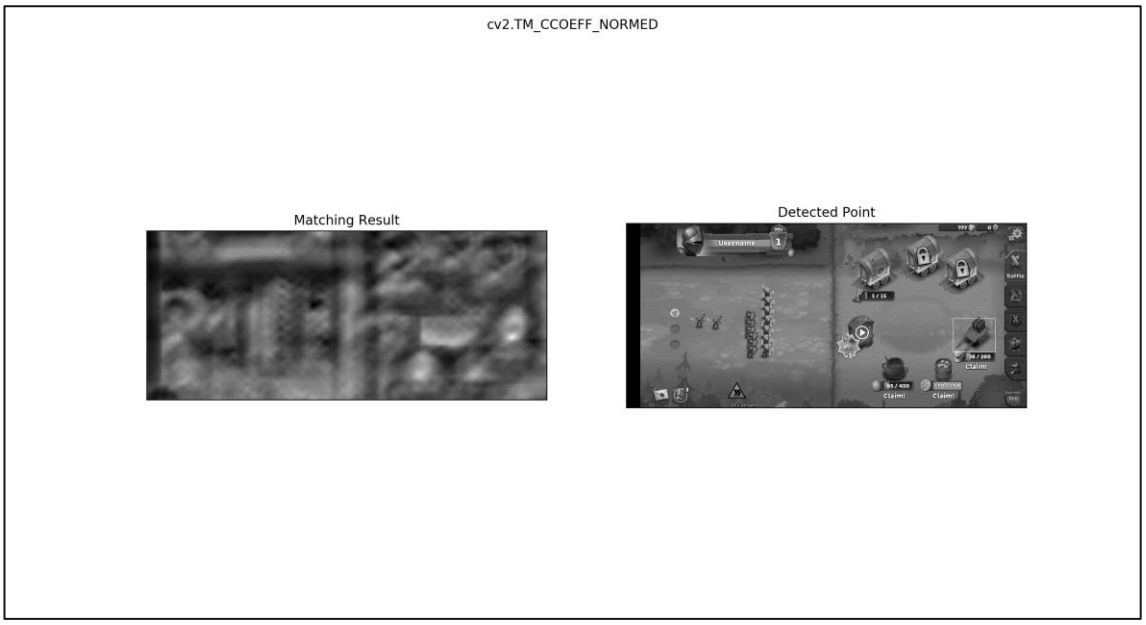
Yksinkertaisesti visualisoituna kuvien ominaisuuksia voidaan verrata matchTemplate nimisellä metodilla. Se käyttää avukseen kahta kuvaa liikuttamalla toista toisen päällä ja samalla vertaamalla osumia näiden kuvien välillä (OpenCV, Feature Matching). Tämä metodi antaa jokaiselle kuvan kohdille tai pikseleille oman arvon riippuen sitä ympäröivien pikselien samankaltaisuudesta verrattaessa mallikuvaan. Funktiossa käytetystä algoritmistä riippuen kaikki kuvan osuvuudet ilmoitetaan tuloksissa asteikolla nollasta yhteen tai yhdestä nollaan.

```
result = cv2.matchTemplate(img_gray, template_rgb, cv.TM_CCOEFF_NORMED)
```

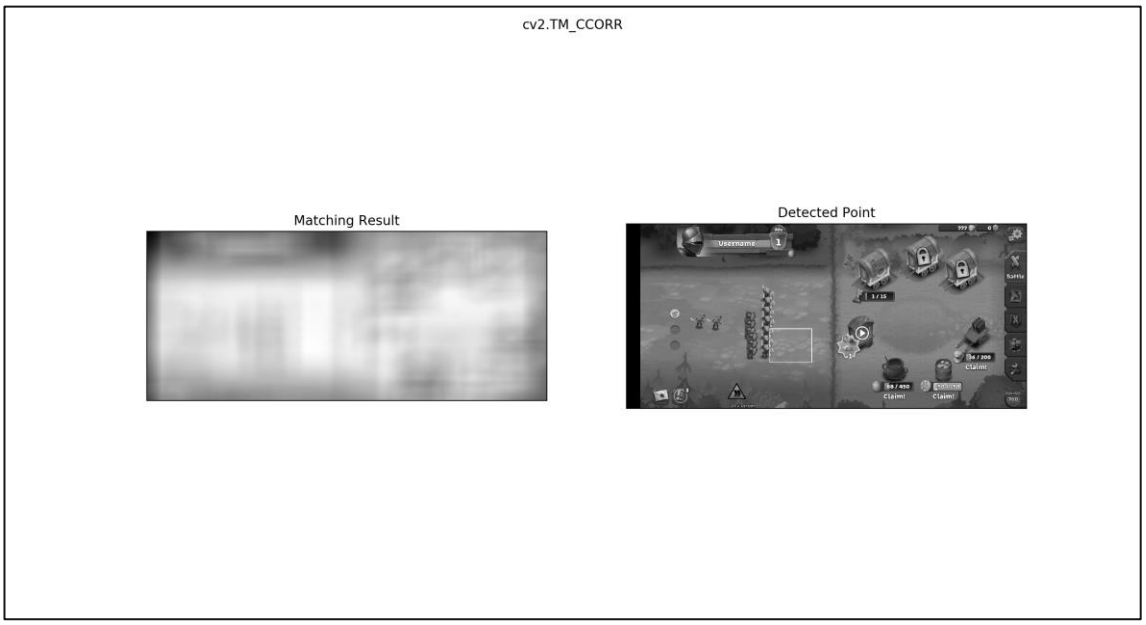
Funktioon annetaan molemmat kuvat muuttujina ja lopuksi määritellään mitä laskutapaa käytetään. Korrelaatiokartta tallennetaan lopuksi muuttujaan result. OpenCV tarjoaa kuusi erilaista tapaa laskea kuvien korrelaatio. Kuvat 7-12 havainnollistavat millaisia tuloksia saadaan eri metodeilla etsittäessä kuvaa 6 laitteen ruudunkaappauksesta.



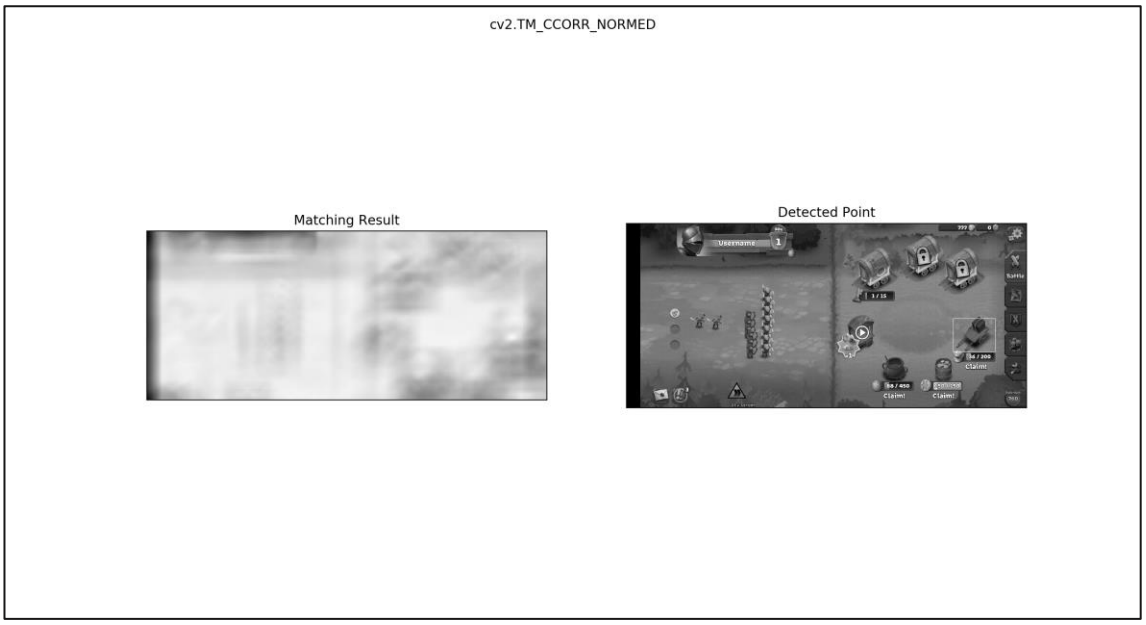
Kuva 6. Mallikuva



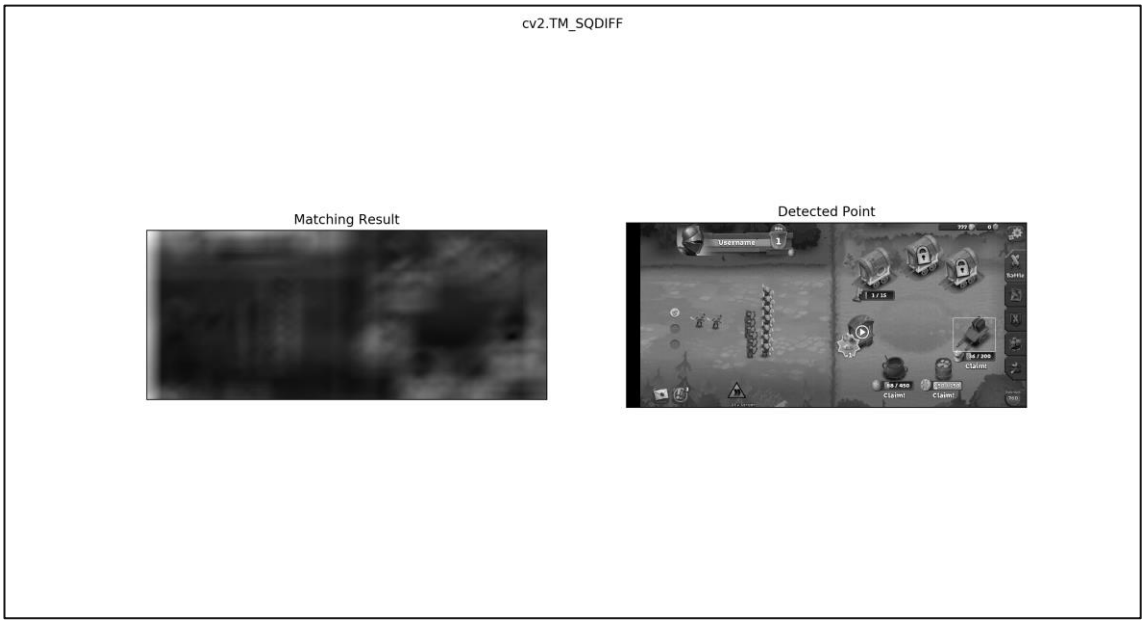
Kuva 7.Tulokset käyttäen CV_TM_CCOEFF_NORMED metodia



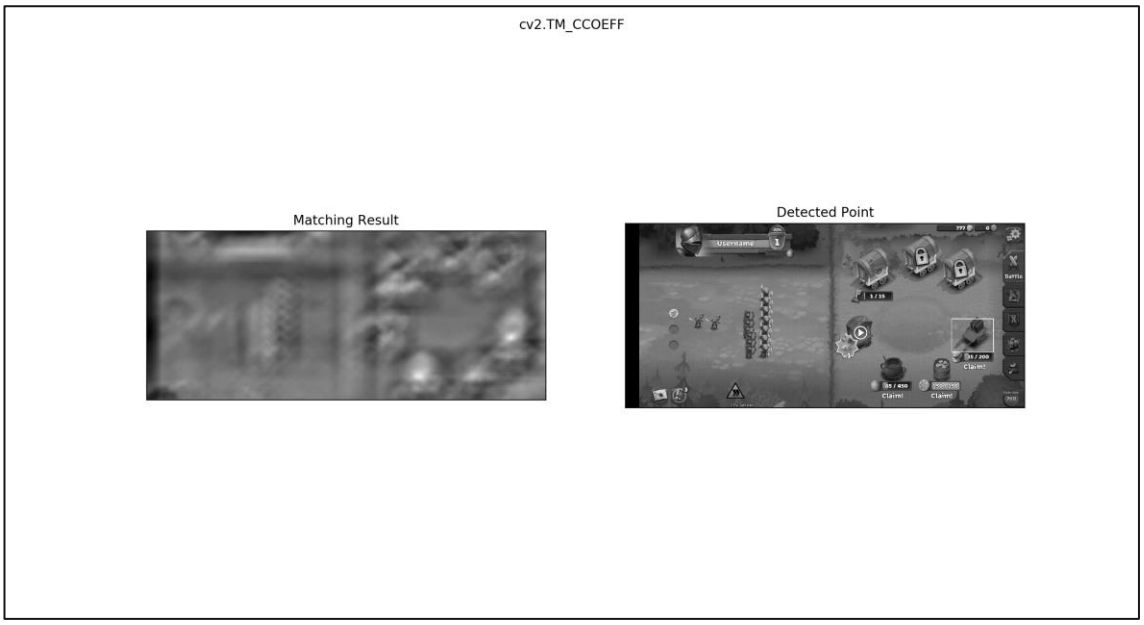
Kuva 8. Tulokset käyttäen CV_TM_CCORR metodia



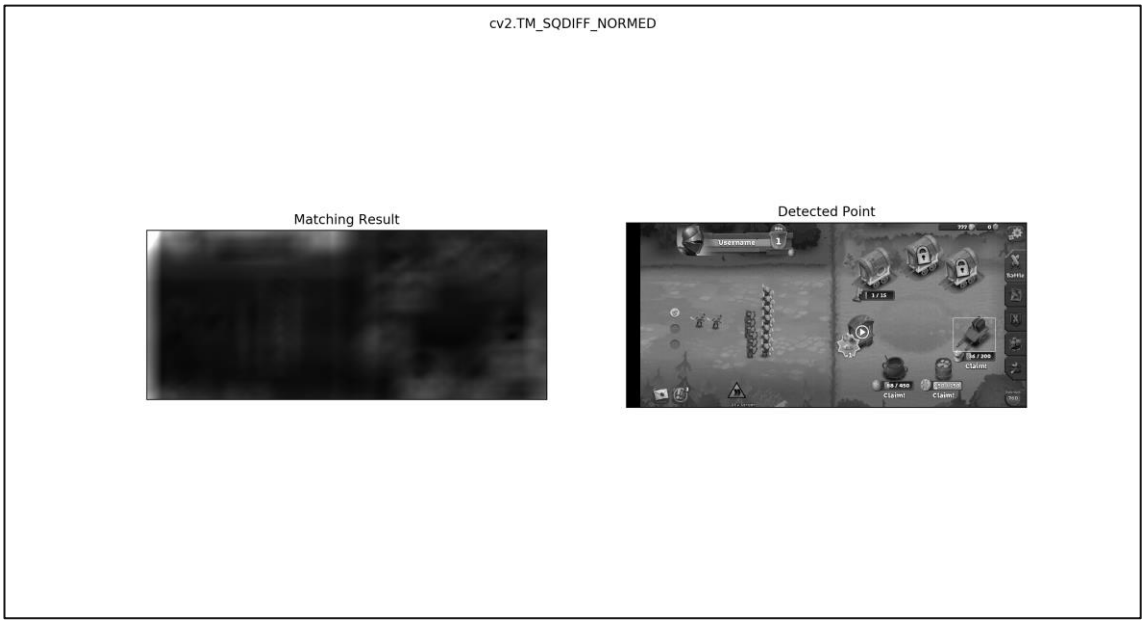
Kuva 9. Tulokset käyttäen cv2.TM_CCORR_NORMED metodia



Kuva 10. Tulokset käyttäen cv2.TM_SQDIFF metodia



Kuva 11. Tulokset käyttäen cv2.TM_CCOEFF metodia



Kuva 12. Tulokset käyttäen cv2.TM_SQDIFF_NORMED metodia

Tuloksista nähdään että vain CV_TM_CCORR -metodin epäonnistui löytää haettava kohde kuvasta. Parhaisiin tuloksiin päästäänkin yhdistelemällä eri metodien tuloksia ja vertaamalla niitä toisiinsa. Virheosumien määrä kuitenkin kasvaa merkittävästi, mikäli kuvassa on enemmän yksityiskohtia.

Kuvan ominaisuudella tarkoitetaan yksinkertaisuudessaan sellaista kuvan osaa, joka on ihmisen tai koneen tunnistettavissa (Tyagi). Ominaisuuksia haetaan rajaamalla kuva tiettyyn alaan ja etsimällä niistä uniikkeja piirteitä. Kuvien

ominaisuudet voidaan määritellä hyviksi ja huonoiksi ominaisuuksiksi sen mukaan, miten helposti ne voidaan erottaa muista kuvan muodoista (OpenCV, Feature Matching). Hyviä ominaisuuksia voivat olla esimerkiksi kuvien kulmat tai joissain tapauksissa jopa pienet pallon muotoiset kuviot. Suurempia määritteleviä muotoja voidaan kuvasta etsiä hakemalla hyviä ominaisuuksia, joita ympäröi suuri määrä visuaalista variaatiota.

6.2.1 Brute-force -menetelmä

Kuvia voidaan verrata ominaisuuksien avulla myös ns. Brute-force -menetelmällä, joka etsii vierekkäisiä piirrekuvaajia yhdestä joukosta ja yhdistelee niitä toiseen joukkoon ominaisuuksia (OpenCV, Introduction to SURF). Näiden joukkojen välimatkat lasketaan ja lähimmät niistä palautetaan.

ORB on avainpisteidenhavaitsija-algoritmi, joka kehitettiin patentoimattomana vaihtoehtona SURF- ja SIFT-algorimeille. ORB on Tyagin artikkelin mukaan tarkempi kuin SURF ja toimii lähes kaksi kertaa niin nopeasti kuin SIFT.
ORB-funktion avulla voidaan hakea kuvasta tunnistettavia avainpisteitä, sekä piirrekuvaajia.

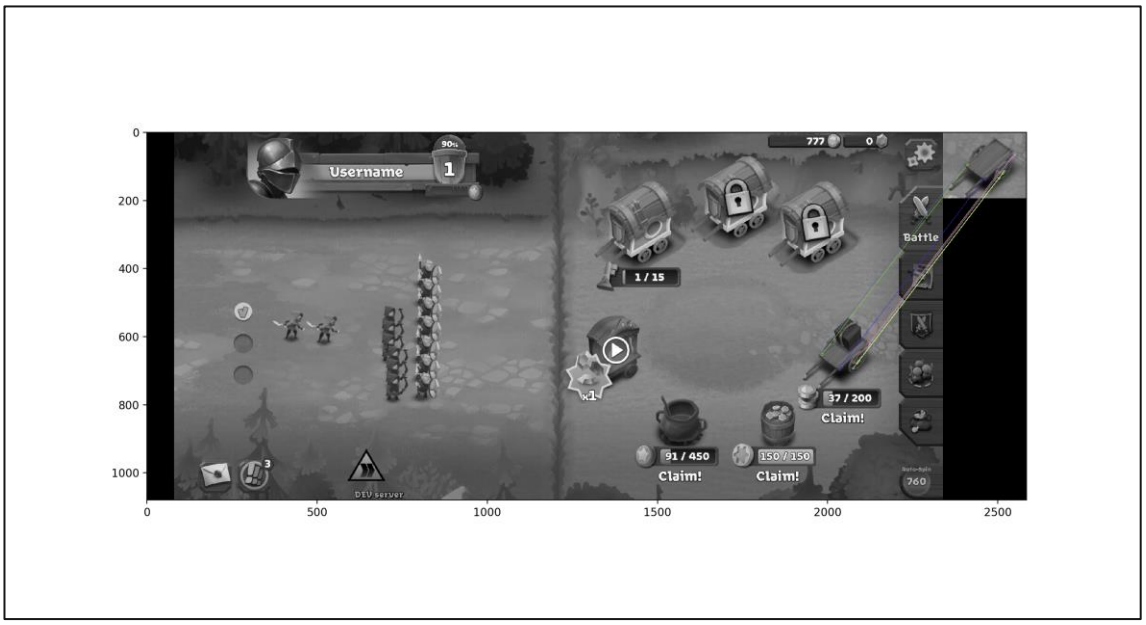
```
orb = cv2.ORB_create(nfeatures = 1000,
scoretype=cv2.ORB_FAST_SCORE)
kp1, des1 = orb.detectAndCompute(img_gray, None)
kp2, des2 = orb.detectAndCompute(template_rgb, None)

bf = cv2.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

matches = bf.match(des1,des2)
```

Ensimmäisessä rivissä ORB alustetaan käyttöä varten. Muuttuja nfeatures määrittää montako ominaisuutta kuvasta etsitään. Tämän jälkeen sitä käytetään tunnistamaan avainpisteet, sekä piirrekuvaajat detectAndCompute-funktiolla. Pisteet ja kuvaajat tallennetaan omiin muuttujiinsa. Tämän jälkeen alustetaan piirrekuvaajia yhdistävä BFMatcher luokka. BFMatcher ottaa parametreikseen normityypin, sekä totuusarvomuuttujan, jolla määritellään käytetäänkö joukkojen ristiintarkistusta vai ei. Normityyppi taas määrittelee, millä menetelmällä välimatkojen mittaus suoritetaan. ORB-algoritmin kanssa on hyvä käyttää

cv.NORM_HAMMING2-mittaustapaa, sen tuottaman binaarisen merkkijonon takia. Lopuksi bf.match palauttaa listan objekteja, joista saadaan eroteltua piirrekuvaajien välimatkat. Lähimmän välimatkan piirrekuvaajilla saadaan tarkimmat tulokset. Kuvassa 13 on visualisoituna molempien kuvien keskenään vertaillut ominaisuudet hyödynnettäessä ORB-algoritmiä. Ominaisuudet on siirretty listaan, josta toisiaan lähimmät tulokset on eritelty. Koska pisteitä on useita, voidaan paikannetun käyttöliittymän objektin sijainti määritellä laskemalla kaikkien pätevien pisteiden keskikohta.



Kuva 13. Osuvimmat tulokset ORB avainpisteiden tunnistajalla

Piirrekuvaajat voidaan hakea vaihtoehtoisesti hyödyntämällä jo aiemmin mainittuja SIFT- tai SURF-algoritmia.

```
sift = cv.xfeatures2d.SIFT_create()
```

SIFT alustetaan xfeatures2d-luokkaa hyödyntämällä, sillä se on patentoimisen myötä siirtynyt maksullisen moduulin alle. Avainpisteiden tunnistaminen käy saman nimisellä kutsulla kuin ORB-esimerkissä, mutta tällä kertaa kutsutaan SIFT-funktiota.

SURF toimii nopeampana vaihtoehtona SIFT-algoritmille ja sitä voidaan ajaa rinnakkain useilla kuvasuhteilla. Alustettaessa SURF-algoritmia voidaan määritellä ehtoja, kuten hessian raja-arvo. Mitä korkeampi arvo, sitä tarkempia tuloksia ja luotettavampia yhtymäkohtia voidaan tuloksissa saavuttaa. (OpenCV, Introduction to SURF)

```
surf = cv2.SURF(400)
```

Aivan kuten aiemmassakin esimerkeissä voidaan myös SURF:n ja SIFT:n avulla hakea avainpisteet sekä piirrekuvaajat ja erotella näistä parhaat osumat.

```
kp1, des1 = sift.detectAndCompute(img_gray,None)
kp2, des2 = sift.detectAndCompute(template_rgb,None)

bf = cv.BFMatcher()
matches = bf.knnMatch(des1,des2,k=2)


good = []
for m,n in matches:
        if m.distance < 0.75*n.distance:
                good.append([m])
```

Lopuksi voidaan luoda lista osumista laskemalla parhaan ja toiseksi parhaan tuloksen suhde ja vertaamalla sitä sopivaan suhdelukuun. Mikäli suhde alittaa annetun luvun lisätään se listaan. ORB-algoritmillä saatu tulos on binäärijonoista koostuva piirrekuvaaja, kun taas SIFT- ja SURF-menetelmillä tämä koostuu liukuluvuista.

6.3 Laitteen kanssa kommunikoiminen

Kun yhdistävien ominaisuuksien koordinaatit on löydetty käyttäen openCV:n tarjoamia funktioita, voidaan Appium-rajapinnan avulla helposti syöttää laitteeseen tai emulaattoriin kosketuksia annettuihin koordinaatteihin. Pisteen paikkaa laskettaessa tulee tapauskohtaisesti ottaa huomioon, että Appium voi lisätä ruudunkaappaukseen mustan kaistaleen, joka on laitteen navigaatiopalkin kokoinen.

Appium-kirjastoa hyödyntämällä voidaan luoda kosketusobjekteja ja antaa niille tapahtumaketju. Kosketusolioon syötettävät tapahtumat ovat nimiltään press, release, moveTo, wait, longPress, cancel ja perform. Appium-dokumentaatiossa suositellaan kosketustapahtumien käyttämiseksi hyödynnettävän sovelluksen elementtejä, mutta tämä voidaan kiertää antamalla kosketus suoraan annettuun koordinaattiin.

```
TouchAction(driver).tap(None, 104, 255, 1).perform()
```

Koska kosketusta ei voida antaa suoraan elementtiin, tulee tap-funktion ensimmäisen kentän olla tyhjä. Tämän jälkeen annetaan koordinaatit, joihin kosketus halutaan antaa ja lopuksi Appium käsketään suorittamaan kosketustoimintoketju. TouchAction-oliolle voidaan tarvittaessa antaa myös moveTo-tapahtuma, jolloin sovellukseen voidaan toteuttaa pyyhkäisy (Appium, Automating mobile gestures, 2019).

7 POHDINTA

Opinnäytetyön tarkoitus oli selvittää soveltuuko kuvantunnistus osaksi mobiilipelien automaatiotestausjärjestelmää. Opinnäytetyön avulla saatiin selville että, tämän kaltainen automaatiotestausjärjestelmä on mahdollista toteuttaa työkaluja hyödyntämällä, mutta se vaatii pidemmän aikavälin ratkaisuna toteuttavalta taholta ylimääräisiä resursseja. OpenCV-ohjelmointikirjasto mahdollistaa kuvantunnistusmenetelmien käytön ja näitä yhdistelemällä saadaan tarvittaessa aikaiseksi riittävät tulokset elementtien paikannukseen. Tarkkojen tulosten saavuttamiseksi kuvantunnistukseen tulee kuitenkin uhrata aikaa ja testauslaitteiston suoritustehoa. Tämän takia menetelmä kärsiikin esimerkiksi reaaliaikaisissa peleissä, joissa elementit saattavat olla aktiivisesti liikkeessä ja sovelluksille tulee antaa syötteitä nopeaan tahtiin. Tämän opinnäytetyön esimerkkeihin kuvantunnistuksen ratkaisut ovat kuitenkin tuottaneet tarkkoja tuloksia.

Opinnäytetyössä käytetty lähestymistapa kärsii myös laitepohjan resoluutioeroihin liittyvistä ongelmista. Kuvantunnistuksen virhealttius kasvaa tilanteessa, jossa testattava peli skaalautuu eri kokoisilla näytöillä eri tavoin. Tämän takia eri resoluution laitteet vaativat aivan oman verrattavista kuvista luodun kuvakirjastonsa. Tämä luo myös ylläpidolle jatkuvasti lisää työtä, kun kuvakirjastoa täytyy päivittää uusien ominaisuuksien myötä. OpenCV-ohjelmointikirjaston toiminnasta on opinnäytetyön avulla saatu lisää tietoa, jolla on kyetty karsimaan heikoimpia menetelmiä ja onnistuttu luomaan pohjaa jatkokehitykselle. Kuvantunnistuksen menetelmät on saatu toimimaan, mutta niiden optimoiminen vaatii jatkotutkimuksia. Yhtä kuvantunnistusmenetelmää, joka toimisi kaikissa tilanteissa ja vastaisi Unity-pelimoottorin haasteisiin ei ole onnistuttu löytämään. Yhdistelemällä eri menetelmiä voidaan kuitenkin päästä lähemmäksi tarkkoja tuloksia.

Tämänkaltaisen ohjelmakokonaisuuden itse rakentamisessa kannattaa ottaa huomioon työn tuottama vaiva ja siitä aiheutuvat kustannukset. Ennen kaikkea samassa tilanteessa tulisi arvioida järjestelmän soveltuvuus omien sovellusten testaamiseen. Vaikka kuvantunnistusta hyödyntämällä on saatu hyviäkin

tuloksia, pyritään jatkossa järjestelmän tiimoilta luomaan luotettavampi menetelmä objektien sijaintien lähettämiseksi testiskriptien käyttöön. Vaihtoehtoinen tapa toteuttaa automaattisia UI-testejä on esimerkiksi Unityn Assets Storen kauttakin ladattava Robert Poienarin kehittämä AltUnityTester työkalu. AltUnityTesterissä on tuki Appium-toiminnoille ja testit voidaan kirjoittaa Javalla tai Pythonilla (Poienar). AltUnityTester on myös mahdollista liittää toimimaan Appium-palvelimen kanssa. Toinen työläämpi vaihtoehto on kirjoitaa itse peli lähettämään tietoa painikkeiden sijainneista Android-rajapinnan kautta.

LÄHTEET

Appium. Automating mobile gestures. Luettu 12.8.2019. http://appium.io/docs/en/writing-running-appium/touch-actions/

Appium. Introduction to Appium. Luettu 22.7.2019. https://appium.io/docs/en/about-appium/intro/

Charles, P. Schultz, R.B. 2017. Game Testing All in One. Dulles, Virginia: Mercury Learning & Information.

Kasurinen, J.P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo Oy.

OpenCV. Feature Matching. Luettu 24.7.2019. https://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html

OpenCV. Introduction to SURF. Luettu 26.7.2019. https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html

OpenCV. OpenCV documentation index. Luettu 20.6.2019. https://docs.opencv.org/

OpenCV. Understanding Features. Luettu 24.7.2019. https://docs.opencv.org/3.1.0/df/d54/tutorial_py_features_meaning.html

Poienar, R. 2019. AltUnityTester. Luettu 15.9.2019. https://assetstore.unity.com/packages/tools/utilities/altunitytester-112101

PyTest. 2004-2017. Luettu 3.6.2019. Full pytest documentation. https://docs.pytest.org/en/latest/contents.html#toc

PyTest. 2015-2019. Luettu 3.6.2019. https://docs.pytest.org/en/latest/capture.html

Tyagi, D. 2019. Introduction To Feature Detection And Matching. https://medium.com/software-incubator/introduction-to-feature-detection-and-matching-65e27179885d