Jere Orava

Mobiilipelien automaattinen suorituskykytestaus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

3.5.2023

Tiivistelmä

Tekijä:	Jere Orava
Otsikko:	Mobiilipelien automaattinen suorituskykytestaus
Sivumäärä:	42 sivua + 4 liitettä
Aika:	3.5.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Opinnäytetyönä tehtiin tapaustutkimus, joka toteutettiin työn ohella eräässä peliprojektissa. Työn aikana työskenneltiin mobiilipelien suorituskykytestauksen parissa. Työkuvaan kuului uusien testien rakentaminen, olemassa olevien testien optimointi, työkalujen ja prosessien laajentaminen sekä koko suorituskykytestauskokonaisuuden tutkiminen. Työn tarkoituksena oli kerätä yleishyödyllisiä oppikokemuksia ja parhaita käytänteitä mobiilipelien suorituskykytestauksessa Unreal Engine 4 -pelimoottorissa.

Tutkimuksessa saatiin kattava näkökulma mobiilipelien suorituskykytestaukseen ja siihen, miten Unreal Engine 4 -pelimoottorin tarjoamat työkalut auttavat testauksessa. Lisäksi saatiin kokemusta ja oppeja siitä, minkälaisiin eri ongelmiin suorituskykytestauksessa saattaa törmätä ja miten ne kannattaa ratkaista.

Unreal Engine 4 osoittautui erittäin käytännölliseksi suorituskykytestien toteuttamiseen, sillä se tarjoaa valmiit työkalut muun muassa profilointiin ja testitulosten visualisointiin. Lisäksi sen alitasopohjainen työnkulku on erittäin sopivaa testien rakentamiseen, sillä testien sisältö voidaan eristää testien omiin alitasoihin ja ladata sisään hallitusti testien aikana.

Isoimmiksi tapaustutkimuksen paljastamiksi ongelmiksi osoittautuivat pelaajan aktivoimat tapahtumat ja satunnaisesti generoidut tasot. Nämä ongelmat vaikeuttivat testien toteuttamista ja testituloksien pitämistä yhtenäisinä oikean pelin pelaamiseen verrattuna.

Suorituskykytestauksen parhaiksi käytänteiksi voidaan tiivistää seuraavat käytänteet: Suorituskykytestien ympäristö on paras luoda vastaamaan mahdollisimman paljon oikean pelin pelaamista. Mikäli pelissä on satunnaisesti generoitua sisältöä, on paras luoda testit kaikista vaativimmalle mahdollisen sisällön permutaatiolle ja testata näin sisällön alin mahdollinen suorituskyky. Pelin tukemille puhelinmalleille tarvitaan tarkat määritelmät, ja tuetut laitteet tulee testata läpikotaisin osana automatisointiputkea. Mikäli suorituskykyongelmia havaitaan, ne tulee korjata välittömästi optimoimalla sisältöä käyttämään laiteresursseja tehokkaammin.

Avainsanat: Unreal Engine, testausautomaatio, mobiilipelien testaus, suorituskykytestaus

Abstract

Author:	Jere Orava
Title:	Automated performance testing of mobile games
Number of Pages:	42 pages + 4 appendices
Date:	3 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisor:	Miikka Mäki-Uuro, Senior Lecturer

This thesis is a case study, which was conducted alongside mobile performance testing work in a professional game project. The scope of work included building new tests, optimizing existing tests, extending tools and processes, and exploring the project's performance testing framework. The purpose of the work was to collect general lessons learned and best practices for mobile game performance testing in the Unreal Engine 4 game engine.

The study provided a comprehensive perspective on mobile game performance testing and how the tools provided by the Unreal Engine 4 game engine help with testing. It also provided experience and lessons learned about the different problems that can arise in performance testing and how to solve them.

Unreal Engine 4 proved to be a very practical tool for performance testing, providing ready-to-use tools for profiling and visualization of test results, among other things. In addition, its sublevel workflow is very suitable for building tests, as test content can be isolated into its own test sublevels and loaded in a controlled way during testing.

The major problems revealed by the case study were player action-powered events and randomly generated levels. These problems made it difficult to run the tests and to keep the test results consistent with that of the real game's performance.

The following practices can be summarized as best practices for performance testing: It is best to create an environment for performance testing that is as close as possible to playing a real game. If the game has randomly generated content, it is best to create tests for the highest possible permutation of the content and thus test the lowest possible performance of the content. The phone models supported by the game need to be precisely defined, and the supported devices should be thoroughly tested as part of the automation pipeline. If performance problems are identified, they should be corrected immediately by optimizing the content to make smarter use of hardware resources.

Keywords: Unreal Engine, automated testing, mobile game testing, performance testing

Sisällys

Lyhenteet

QA:	Quality assurance. Laadunvarmistus.
UE:	Unreal Engine. Epic Games:n pelimoottori, josta on jo viisi versiota (UE, UE2, UE3, UE4 ja UE5).
UDK	Unreal Development Kit. Unreal Engine -pelimoottorin kehityspaketti.
UAT	Unreal Automation Tool. Unreal Engine -pelimoottorin automaattinen testaus- ja rakennustyökalu.
CSV	Comma-separated values. Taulukkomuotoinen tiedostotyyppi.
SVG	Scalable vector graphics. Kaksiulotteisten vektorikuvien tiedostotyyppi.
APK	Android package. Android-sovellusohjelmien pakettitiedosto.
SDK	Software development kit. Ohjelmistokehityspaketti.
NDK	Android native development kit. Android-kehityspaketti.
JDK	Java development kit. Java-kehityspaketti.
CI	Continuous integration. Ohjelmistotuotannon menetelmä, jossa ohjelmistomuutosten tapahtuessa muutokset integroidaan automaattisesti projektin versionhallintaan.
AWS	Amazon Web Services. Amazonin pilvipalvelualusta.
iOS	iPhone Operating System. Applen kehittämä käyttöjärjestelmä.

1 Johdanto

Pelialan suuret yritykset ovat alkaneet nähdä mobiilialustojen tarjoaman mahdollisuuden jatkaa pelaamista missä ja milloin vain. Modernit ison budjetin pelit tehdään tai käännetään yhä useammin mobiililaitteille tehokkaampien puhelimien ja etenkin ristiin pelaamisen suosion kasvun ansiosta.

Tietokone- ja konsolipelien kääntäminen mobiilialustoille on iso ja hankala tehtävä, joka vaatii kokeneita pelinkehittäjiä ja paljon erilaista teknistä osaamista. On projekteja, joissa mobiililaitteille kääntäminen tehdään vasta PC- tai konsoliversion jälkeen, ja on projekteja, joissa mobiiliversio kehitetään PC- tai konsoliversion rinnalla. Molemmissa kehitystavoissa tarvitaan tiukkaa laaduntarkkailua ja suorituskykytestausta etenkin mobiiliversion puolella.

Tämän työn tarkoituksena on toimia tapaustutkimuksena suuren peliprojektin automatisoidun suorituskykytestauksen parissa tehdystä työstä, jossa toteutetaan suorituskykytestejä ja parannellaan olemassa olevia testaustyökaluja ja -prosesseja. Tavoitteena on kerätä yleishyödyllisiä oppikokemuksia ja parhaita käytänteitä suorituskykytestauksessa Unreal Engine 4 -pelimoottorissa.

Yritys, jossa projektia tehdään, on suomalainen mobiilipelien kehittäjä ja julkaisija, joka kehittää pelejä keskisuurille käyttäjäryhmille. Yrityksessä on noin 60 työntekijää. Yhtiön omien peliprojektien painotus on ilmaiseksi pelattavissa mobiilipeleissä, mutta se tarjoaa kehitysapua palvelumuodossa kaikentyyppisiin peliprojekteihin, joissa käytetään Unity- tai Unreal Engine -pelimoottoreita.

Peliprojekti, jossa suorituskykytestausta tehdään, on julkistamaton peli ja siksi toistaiseksi salainen. Opinnäytetyössä keskitytäänkin tästä syystä vain projektissa opittuihin asioihin järjestelmätasolla ja jätetään pelin yksityiskohdat mainitsematta. Esimerkkitapaukset on korvattu yleismuotoisilla tapauksilla.

Opinnäytetyön menetelmäksi valittiin tapaustutkimuksen kehys. Opinnäytetyöraportti kirjoitettiin peliprojektissa työskentelyn ohella ja siinä tutkitaan, miten

mobiilipelien suorituskykytestaus tehdään projektissa ja millaisia ongelmia suorituskykytestausta tehdessä kohdattiin. Samalla pohditaan, mitä parhaita käytänteitä projektista opittiin.

Raportti aloitetaan käymällä läpi, mitä ohjelmistotestaus on yleisesti ja miten pelien testaus eroaa siitä. Tämän jälkeen tutkitaan, mitä manuaalinen testaus ja testausautomaatio ovat ja mitkä niiden vahvuudet ja heikkoudet ovat.

Toteutusosassa syvennytään Unreal Enginen tuomiin työkaluihin ja suorituskykytestauksen toteutuksen perustaan. Siinä käydään läpi, miten projektin suorituskykytestit toteutettiin ja miten työkaluja kehitettiin opinnäytetyön aikana, sekä katsotaan, miten testitulokset kerätään ja miten testejä voidaan ajaa Android-puhelimilla.

Lisäksi työssä mietitään testiajojen automatisointia jatkuvan integraation avulla ja pohditaan sen ratkaisemia ongelmia. Viimeisessä osiossa käydään läpi, mitä ongelmia projektissa kohdattiin ja kerätään niistä ongelmakohtaiset parhaat käytänteet.

2 Ohjelmistotestaus

Ohjelmistotestaus on ohjelmistokehityksen tärkeä osa, jolla varmistetaan, että ohjelmisto toimii, niin kuin se on suunniteltu. Sillä voidaan myös varmistaa, että sovellus täyttää kaikki liiketoiminnan vaatimukset jokaisen muutoksen jälkeen.

Ohjelmistotestausta on paljon eritasoista, ja sen toteutustapoja on monia. Tässä luvussa käydään läpi, mitä ohjelmistotestaus käytännössä on, miksi sitä kannattaa harjoittaa ja miten sitä tarkalleen ottaen erilaisissa moderneissa ohjelmistoyrityksissä tehdään.

Pohjimmiltaan ohjelmistotestausta tehdään, jotta voidaan vähentää epävarmuutta ja tietämättömyyttä ohjelmiston laadusta ja heikkouksista. Kun tieto ohjelmiston laadusta ja heikkouksista on saatu, voidaan tehdä paremmin tiedostettuja päätöksiä kehityksen eri vaiheissa. Toisin sanoen ohjelmistotestaus auttaa tiedostamaan riskejä ohjelmistokehityksen aikana. [1.]

Osana ohjelmistotestausta on laadunvarmistus, jossa keskitytään varmistamaan, että lopullinen ohjelmisto täyttää käyttäjien laatuvaatimukset, kuten tarpeeksi nopeat latausajat ja hyvä käyttöliittymän reagoivuus. Se siis pyrkii saamaan kehittäjiä parantamaan ohjelmiston laatua tarvittaessa kohtaamaan laatuvaatimukset. Laadunvarmistukselle olennaisinta on se, että sitä ei tehdä vasta lopuksi, vaan jatkuvasti, jotta voidaan varmistaa, että laatu pysyy tasaisesti hyvänä. Tämä on tärkeää, sillä virheiden korjaus tulee kalliimmaksi ajan myötä. Kehitysprosessit sekä muutosten arviointi ja testaaminen mahdollisimman lähellä lähdettä takaavat, että virheet huomataan ajoissa ja että ne eivät pääse kasvamaan suuremmiksi ongelmiksi. [1.]

2.1 Ohjelmiston testaus

Perinteisen ohjelmiston testauksessa keskipisteenä on toiminnallisuuksien ja käyttöliittymän testaus. Toiminnallisuuden testaus voidaan jakaa kolmeen eri osa-alueeseen:

- ysikkötestaus
- integraatiotestaus
- hyväksyntätestaus.

Yksikkötestauksessa keskitytään yhden komponentin eli luokan tai jonkin tietyn funktion toimintaan. Sillä tarkistetaan, toimiiko ohjelmiston jokin tietty osa eristetyssä ympäristössä. Integraatiotestauksessa testataan useampaa komponenttia yhdessä ja tarkistetaan, toimivatko ne yhdessä oletetulla tavalla. Hyväksyntätestauksessa testataan ohjelmistoa käyttäjän näkökulmasta ja tarkistetaan, täyttääkö ohjelmiston jonkin tietyn osan toiminnallisuus käyttäjän vaatimukset. Kun kaikki ohjelmiston hyväksyntätestit on ajettu ja tulokset ovat hyväksyttäviä, ohjelmisto voidaan antaa loppukäyttäjien testattavaksi. [2.]

2.2 Pelien testaus

Pelien testauksessa otetaan mallia perinteisen ohjelmiston testauksesta, mutta se eroaa siitä usealla eri tavalla pelien graafisten eroavaisuuksien vuoksi.

Esimerkiksi audiovisuaalinen testaus on yleisempää peleissä kuin perinteisissä sovelluksissa. Audiovisuaalisella testauksella testataan, näyttääkö ja kuulostaako peli siltä, mitä oletetaan. Siinä katsotaan ja kuunnellaan esimerkiksi, puuttuuko ympäristöstä tekstuureja tai ääniä tai onko visuaalisessa ilmeessä tai äänimaailmassa jotain pelikokemusta häiritsevää. Tämä on hyvin subjektiivinen testauksen osa-alue, joten sitä ei usein voi toteuttaa muu kuin ihminen. Tosin joissain yksinkertaisissa tapauksissa sitä voidaan automatisoida ruudunkaappauksien tai videon ja äänen tallennuksen avulla, minkä jälkeen jokin testausautomaatiotyökalu voi vertailla tuloksia. [3.]

Toisena esimerkkinä on suorituskyvyn testaus, joka on myös tämän opinnäytetyön pääaihe. Suorituskyvyn testauksessa mitataan, miten jokin tietty pelin taso, välivideo tai animaatio suoriutuu eri laitteilla ja asetuksilla. Tärkeinä mittausyksikköinä käytetään usein ruudunpäivityksiä sekunnissa (FPS, frames per

second) ja muistinkäyttöä (MB, megabyte). Lisäksi verkossa pelattavissa peleissä testataan verkkoyhteyden viiveen ja pakettihävikin vaikutusta pelaamiseen. Suorituskyvyn testaus on tärkeää etenkin PC- ja mobiilialustoilla, sillä erilaisia kokoonpanoja ja laitemalleja on lukemattomia.

Suorituskyvyn testausta tehdään myös perinteisessä ohjelmistokehityksessä, mutta se eroaa pelien suorituskykytestauksesta monin pienin tavoin. Esimerkiksi ruudunpäivitys ja liika muistinkäyttö eivät ole usein ongelma sovelluksissa samalla tavoin kuin peleissä. Toisaalta verkon ja palvelinten kuormituksen kesto ovat tärkeitä tarkkailun kohteita samalla tavoin kuin pelikehityksessä. [4.]

2.3 Manuaalinen testaus

Manuaalinen testaus on testausta, jossa ihminen on aktiivisesti osallisena. Se vaatii analyytikoita ja laadunvarmistusinsinöörejä, jotka suunnittelevat, luovat ja toteuttavat testejä sekä keräävät ja analysoivat niiden tuloksia [1].

Manuaaliset testausprosessit ovat usein hitaita, mutta joustavia. Manuaalinen testaus sopii hyvin monenlaiseen luovaan testaukseen, kuten pelin pelituntuman ja audiovisuaalisen puolen toimivuuden testaamiseen. Monissa pelistudioissa manuaalinen testaus onkin yleisin tapa testata kaikki, mitä pelissä on, jolloin automaation hyödyntäminen jää vähälle. Vaikka manuaalinen testaus ei aina olisikaan paras vaihtoehto kaikkeen testaukseen, tähän ilmiöön on yleensä hyvät syyt, kuten se, että testaus on helppo ulkoistaa toisen yrityksen tehtäväksi.

Manuaalinen testaus ei kuitenkaan ole toimiva tapa testata ohjelmiston komponenttien toimivuutta. Toisin sanoen sillä ei saada tietoa ohjelman sisäisistä virheistä, jotka eivät ilmene ilmiselvästi näytöllä, mutta saattavat tuottaa ongelmia ajan mittaan.

2.4 Testausautomaatio

Testausautomaatiolla tarkoitetaan testauksen automatisointia, eli toisin sanoen automaattisesti ajettavia testejä, jotka tietokone suorittaa. Testi kirjoitetaan ensin jollakin ohjelmointikielellä, minkä jälkeen testin voi ajaa joko nappia painamalla tai automaattisesti toisen automaatiotyökalun toimesta. Automaattiset testit ovat yleensä hyvin nopeita ajaa ja tarkkoja tuloksissaan, ja ne säästävät paljon aikaa manuaalisesta testauksesta, jolloin testaajilla jää aikaa muuhun, luovempaan työhön. Niillä voidaan korvata suoraviivaiset testit, kuten yksikkötestit, integraatiotestit ja hyväksyntätestit, joissa ei tarvita ihmisen luovuutta tai pragmaattisuutta. Testausautomaatio on myös erinomainen tapa toteuttaa suorituskykytestejä, sillä se suorittaa testauksen täsmällisesti joka kerta. Näin ollen myös testitulokset ovat hyvin tarkkoja ja luotettavia.

Testausautomaatio ei kuitenkaan voi täysin korvata manuaalista testausta. Syitä tähän on monia, kuten liian monimutkaiset tai arvaamattomat testitapaukset ja niistä johtuvat satunnaiset väärät negatiiviset tulokset. Automaattinen testaus ei myöskään kannata testitapauksissa, jotka ovat harvinaisia tai jatkuvasti muuttuvia, sillä niiden jatkuva ylläpito tulee kalliiksi. [5.]

Testausautomaatio ei ole ilmaista. Sen rakentaminen vie aloitettaessa paljon resursseja, ja sitä pitää jatkuvasti ylläpitää. Muutokset testattaviin komponentteihin tai uudet peliin lisätyt toiminnallisuudet saattavat rikkoa testejä, jolloin testejä täytyy päivittää ottamaan muutokset huomioon. Näin ollen kaikkea ei ole järkevää automatisoida, vaan on suositeltavaa arvioida automaation tarve tarkkaan tapauskohtaisesti. [6.]

3 Suorituskyvyn testaus UE4-pelimoottorissa

Opinnäytetyön tapaustutkimuksen pohjana toimivassa projektissa on käytössä Unreal Engine 4 (lyh. UE4), joka on Epic Gamesin vuonna 2015 julkaisema pelimoottori. Epic Games tunnetaan sellaisista peleistä kuin Unreal, Gears of War, Bulletstorm ja Fortnite. Kaikki Epicin pelit on kehitetty Unreal Engine -pelimoottorilla, ja se on kehittynyt vuosien saatossa paljon.

Unreal Engine 3 (lyh. UE3) oli ensimmäinen versio Epicin pelimoottorista, josta julkaistiin ilmainen versio nimeltä Unreal Development Kit (lyh. UDK) kaikkien saataville vuonna 2004.

Vuonna 2014 julkaistu Unreal Engine 4 (lyh. UE4) korvasi UE3:n käyttämän UnrealScript-ohjelmointikielen C++-ohjelmointikielellä sekä lisäsi reaaliaikaisen valaistuksen ja paljon muuta. [7.]

Unreal Engine -pelimoottorin uusin versio, Unreal Engine 5, julkaistiin vuonna 2022 ja on siitä lähtien saavuttanut suurta suosiota sen tuomilla Lumen- ja Nanite-teknologioilla, jotka helpottavat valaistuksen ja 3D-mallien käyttöä peliprojekteissa samalla niiden suorituskykyä parantaen [8].

Monet peliprojektit ovat vaihtamassa UE4:stä UE5:een sen tuomien parannuksien vuoksi, mutta tämän opinnäytetyön tutkimassa pelissä käytetään toistaiseksi vielä Unreal Engine 4:ää. Unreal Engine 5:n tuomat ominaisuudet auttavat huomattavasti mobiilipelien optimoinnissa ja valaistuksen rakentamisessa, joten vaihdos tehdään varmasti hyvin ennen pelin lopullista julkaisua.

3.1 Unreal Automation Tool -kehys

Unreal Automation Tool (lyh. UAT) on UE4:n sisäinen automaatiotestauksen kehys, joka mahdollistaa testien lisäämisen UE4:n kehitysympäristöön ja niiden ajamisen.

UAT:n käyttöliittymä on yksinkertainen. Suurin osa ajasta näkymää ei käytetä muuhun kuin testien ajamiseen. Tämä onnistuu valitsemalla testit, jotka halutaan ajaa, ja painamalla nappia käyttöliittymästä.

Uuden testin luonti UAT-käyttöliittymään onnistuu luomalla C++-tiedosto ja kirjoittamalla sen sisään makro, joka merkitsee testin luokan ja nimen UAT-kehykseen. Makron parametreiksi tarvitaan testiluokan nimi, testin nimi ja testin asetukset, kuten esimerkkikoodissa 1.

```
IMPLEMENT_SIMPLE_AUTOMATION_TEST(FMyUnitTestRunner, "MyUnitTest",
     EAutomationTestFlags::EditorContext
     | EAutomationTestFlags::ProductFilter)
```

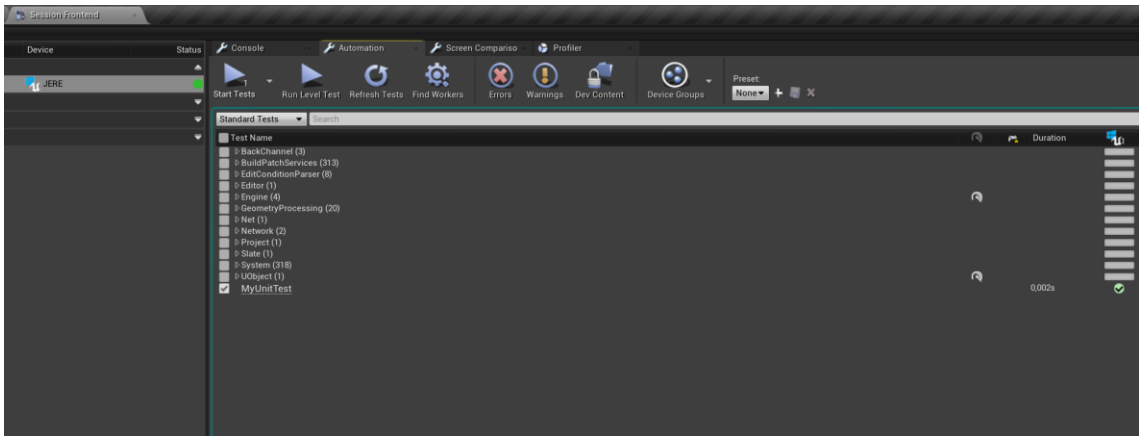
Esimerkkikoodi 1. Esimerkki makrosta, jolla testi merkitään UAT-kehykseen.

Makron lisäksi testi tarvitsee logiikkaa RunTest-funktion muodossa, joka ottaa parametrikseen merkkijonon osoitteen. Testin sisällä voidaan käyttää useampaa eri tapaa tarkistaa testin onnistuminen, mutta yleisin tapa tehdä tarkistus on UAT:n TestTrue-funktiolla, kuten esimerkkikoodissa 2. [9.]

```
/// Esimerkki funktiosta, jonka sisälle voidaan kirjoittaa testilo-
/// giikkaa
bool FMyUnitTestRunner::RunTest(const FString& InParameter)
{
     TestTrue(“My test was successful”, 2 > 1);
     return true;
}
```

Esimerkkikoodi 2. Esimerkki testistä, joka testaa, että numero 2 on isompi kuin numero 1.

Näin ollen testi näkyy UAT:n käyttöliittymässä kuten kuvassa 1, ja se voidaan valita ja ajaa.



Kuva 1. Testin voi nyt ajaa UAT:n käyttöliittymässä, ja se näyttää vihreää.

3.2 Kamerapolkujen rakennus

Jotta suorituskykytestejä voidaan rakentaa, tarvitaan ratkaisu, jolla kameraa voidaan ajaa määriteltyä polkua pitkin. Projektissa oli jo ennen tämän opinnäytetyön aloitusta ratkaisu, jota tässä luvussa käytetään esimerkkinä.

Unreal Engine tarjoaa valmiiksi USplineComponent-nimisen luokan, joka on UActorComponent-luokan toteutus. USplineComponent-luokka tuo mihin vain AActor-luokkaan toiminnallisuuden toimia polkuna, joten sitä voidaan käyttää hyödyksi tekemään kamerapolun toiminnallisuudet yksinkertaisesti ja intuitiivisesti.

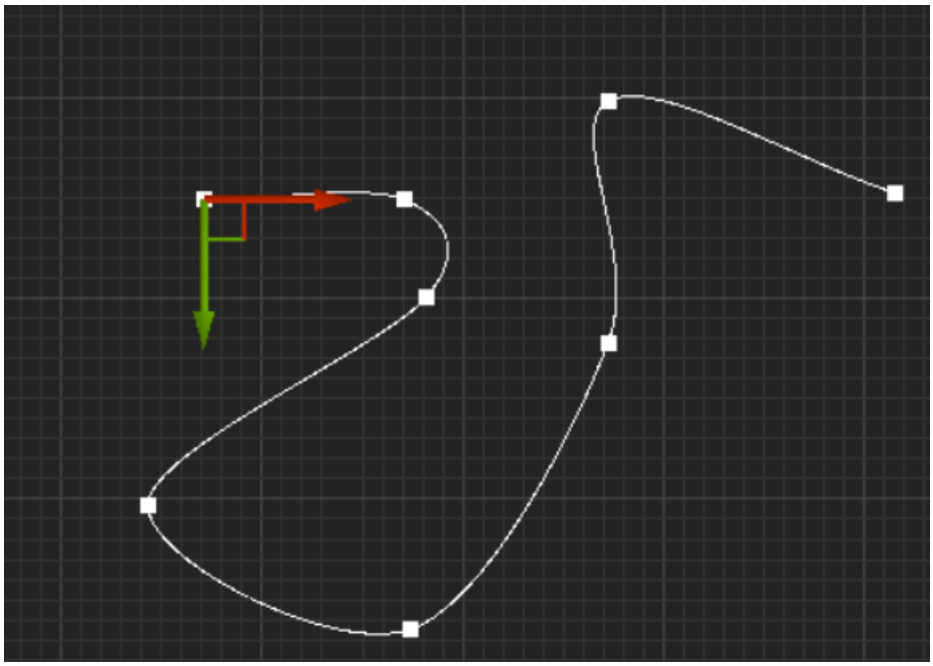
Tätä käyttötarkoitusta varten projektiin oli luotu luokka nimeltä ACameraSpline, joka perii Actor-luokan. ACameraSpline-luokasta löytyy muuttuja nimeltä SplineComponent, joka pitää sisällään osoittimen USplineComponent-tyyppiseen komponenttiin. Esimerkkikoodissa 3 nähdään, miten kamerapolun luokka oli luotu projektissa.

```
private:
     USplineComponent* SplineComponent;

UCLASS()
class ACameraSpline : public AActor
{
…
```

Esimerkkikoodi 3. ACameraSpline-luokan alustus. UCLASS-makro merkitsee luokan UE4-pelimoottorin luokaksi.

Käytännössä luodun kamerapolun muokkaus näyttää kuvan 2 mukaiselta.



Kuva 2. Kamerapolun muokkausnäkymä UE4:n editorissa ylhäältäpäin katsottuna.

Luokka sisältää myös logiikkaa, joka vie kameraa eteenpäin kamerapolulla. Luokasta löytyy tätä varten Tick-funktio, joka ottaa sisäänsä kuluneen delta-ajan. Tick-funktion tehtävänä on pitää kirjaa polulla kuluneesta ajasta. Esimerkkikoodissa 4 nähdään, miten kamerapolun Tick-funktio oli toteutettu projektissa.

```
void ACameraSpline::Tick(const float DeltaSeconds)
{
     CurrentTimeAtSpline += DeltaSeconds;
     CurrentTimeAtSpline = FMath::Clamp(CurrentTimeAtSpline, 0.0f,
          TimeToRunAtSpline);
}
```

Esimerkkikoodi 4. ACameraSpline-luokan Tick-funktio. Aikaa kasvatetaan kuluneen ajan mukaan ja rajoitetaan polulle asetettuun juoksuaikaan.

Lisäksi luokka tarjoaa rajapinnan muille luokille, jolla muut pelin komponentit saavat tietoonsa kamerapolun sijainnin ja suunnan tietyllä ajanhetkellä, sekä milloin kamerapolun aika on loppunut. Esimerkkikoodissa 5 nähdään kamerapolun rajapinnan tärkeimmät funktiot.

```
public:
     bool HasTimeEnded() const;
     FVector GetCurrentPositionAtTime() const;
     FRotator GetCurrentRotationAtTime() const;
```

Esimerkkikoodi 5. ACameraSpline-luokan julkinen rajapinta, jonka kautta muut komponentit saavat tietoa kamerapolun tilasta.

Kun tämä opinnäytetyö aloitettiin, kamerapolkujen ajon kesto oli suoraan määritettävissä editorissa. Kamerapolkuja käytettäessä kuitenkin huomattiin, että koska ajon kesto ja polun pituus ovat suoraan sidoksissa siihen, kuinka nopeasti kamera liikkuu polulla, ei tämä ollut hyvä tapa pitää kamerapolkujen nopeuksia yhtenäisinä. Tästä syystä rajapintaa muutettiin niin, että editorista muokataan ajan sijaan kamerapolun nopeutta. Lisäksi asetettiin kaikille kamerapoluille oletuksena sama nopeus, jota käytetään jollei sitä erikseen ole muokattu kamerapolussa.

Lisäksi huomattiin tarve luoda kamerapolkuja, jotka eivät liikuta kameraa, vaan toimivat niin sanottuina staattisina kamerapaikkoina. Nämä ovat erityisen hyödyllistä tarkkailemaan pelin erilaisia tapahtumia tai animaatioita ilman, että kameran liikkuminen vaikuttaa suorituskykyyn. Tätä varten luotiin staattisen kamerapaikan lisätoiminnallisuus, jonka voi kytkeä päälle editorista yksittäisille

kamerapoluille. Tällöin kamera ei liiku, vaan seisoo paikoillaan editorissa määritetyn ajan.

Kamerapolkujen rakennuksen iteroinnin helpottamiseksi lisättiin toiminnallisuus merkitä, mitkä kamerapolut testit voivat löytää. Tämä oli tärkeä lisäys, sillä se toi kyvyn ajaa testejä vain niillä kamerapoluilla, joihin on tehty muutoksia, jotta näitä muutoksia voitiin testata nopeasti. Viisi minuuttia kestävä testi voitiin näin ajaa alle minuutissa ja nähdä vain muokattujen kamerapolkujen uusi käyttäytyminen.

Tätä kamerapolku-luokkaa hyödynnetään tämän opinnäytetyön suorituskykytestauksessa, jossa kamera liitetään kamerapolkuun ja sen liikkuessa kamera kerää profilointidataa suorituskyvystä. Liitteessä 1 on esimerkki ACameraSpline.h-luokasta.

3.3 Suorituskyvyn profilointi

Jotta kamera-ajoista saadaan irti dataa laitteen suorituskyvystä, tarvitaan jonkintyyppinen profilointityökalu. Unreal Engine 4 tarjoaa tähän sisäänrakennetun ratkaisunsa luokalla nimeltä FcsvProfiler, jota myös tässä projektissa käytettiin.

FCsvProfiler-luokan toiminnallisuus on hyvin yksinkertaista. Luokassa on funktiot BeginCapture ja EndCapture. Kun BegineCapture-funktiota kutsutaan, profiloija aloittaa profiloinnin. Profiloinnin aikana profiloija pitää kirjaa muun muassa ruutujen määrästä ja muistinkäytöstä. Kun EndCapture-funktiota kutsutaan, profiloija lopettaa profiloinnin ja tallentaa kerätyn profilointidatan ennalta määritettyyn CSV-tiedostoon.

Lisäksi UE4 tarjoaa sisäänrakennetun OnHitchDetectedDelegate-delegaatin, johon voidaan liittää omia funktiokutsuja. Tällä delegaatilla saadaan tietoon testin aikana tapahtuneet jäätymiset ja se, millä ruudunpäivityksellä ne ovat tapahtuneet. Jäätymisistä voidaan jopa ottaa ruudunkaappaukset UE4:n

sisäänrakennetulla RequestScreenshot-funktiolla, jotta voidaan myöhemmin tarkkailla, missä kohtaa jäätymiset tapahtuivat ja mitä ruudulla sillä hetkellä näkyi.

4 Komentomalli ja testikomennot

Komentomalli on ohjelmoinnin suunnittelumalli, jossa koodissa rakennetaan useamman funktiokutsun sisältämä jono ja ajetaan ne yksi toisensa jälkeen. Komennosta toiseen siirrytään heti, kun aikaisempi komento ilmaisee tehtävänsä tehdyksi. [10.]

Unreal Engine 4:n sisäänrakennettu ADD_LATENT_AUTOMATION_COMMAND-makro käyttää komentomallia ja mahdollistaa testien ajamisen useammassa eristetyssä vaiheessa. Se ottaa sisäänsä minkä vain luokan, joka toteuttaa IAutomationLatentCommand-rajapinnan, ja kutsuu luokan Update-funktiota automaattisesti, kunnes se palauttaa positiivisen arvon (true). Makro sijaitsee UE4:n AutomationTest-luokassa, joten sitä voidaan käyttää kaikissa testeissä, jotka perivät sen.

Komentojen, jotka toteuttavat IAutomationLatentCommand-rajapinnan, täytyy myös toteuttaa rajapinnan vaatima Update-funktio, jossa komennon logiikka elää.

Yksinkertainen suorituskykytesti voidaan ajatella jaettuna kolmeen eri vaiheeseen:

1. tason lataus
2. alatasojen suoratoisto
3. suorituskykytestin ajo.

Esimerkkikoodissa 6 nähdään ote kolmesta komennosta, jotka toimivat testin eri vaiheina projektin suorituskykytestauksen koodissa.

```
ADD_LATENT_AUTOMATION_COMMAND(FLoadMapCommand(MapPath))
ADD_LATENT_AUTOMATION_COMMAND(FStreamSublevelCommand(SublevelPath))
ADD_LATENT_AUTOMATION_COMMAND(FRunCameraSplinesCommand())
```

Esimerkkikoodi 6. Jono komentoja, jotka toteuttavat suorituskykytestin.

Liitteessä 2 on esimerkki FPerformanceTestRunner.h-luokasta, jossa suorituskykytesti ajetaan ja jossa testit lisätään UAT-kehykseen.

Testien iteroimisen edun mukaista on myös, että testejä voidaan ajaa editorissa. Tätä varten projektissa on vielä yksi komento, joka avaa editorissa peli-ikkunan testejä varten. Se sisällytetään editorissa esimerkkikoodin 7 mukaisesti ennen muita komentoja.

```
if (GEngine->IsEditor())
{
     ADD_LATENT_AUTOMATION_COMMAND(FRequestSessionStartCommand())
}
```

Esimerkkikoodi 7. Komento, joka lisätään suorituskykytestin komentojonoon vain, jos testi ajetaan editorissa.

Komennon logiikka alustaa peli-istunnon editorissa ja odottaa, että se on valmis testejä varten. Liitteessä 3 on esimerkki FRequestSessionStartCommand.h-luokasta, jossa peli-istunto aloitetaan editorin ollessa päällä.

Kaikki edellä mainitut komennot olivat tämän opinnäytetyön alkaessa jo toteutettuna projektissa. Itse suorituskykytestin ajavaan komentoon tuli kuitenkin jatkokehitystä, johon paneudutaan luvussa 4.3 (Komento 3: Kamera-ajo).

4.1 Komento 1: Tason lataus

Tason lataus on usein ensimmäinen vaihe suorituskykytestausta. Sen aikana ladataan testin alla oleva taso ja sen sisältö. Taso ja sen sisältö voi olla mitä vain yhden animoidun olion tai kokonaisen pelattavan tason väliltä. Unreal Engine 4:ssä, taso voidaan ladata helposti sisäänrakennetun UGameplayStatics-luokan OpenLevel-funktiolla. Esimerkkikoodissa 8 nähdään ote OpenLevel-funktion kutsusta, joka aloittaa tason latauksen.

```
bool FLoadMapCommand::Update()
{
     UGameplayStatics::OpenLevel(GWorld, *m_MapPath);
     …
}
```

Esimerkkikoodi 8. OpenLevel-funktion kutsu. Ladattava taso valitaan FLoadMapCommand-komennolle syötetyn nimen perusteella.

Seuraavaksi komento odottaa, että koko taso on ladattu ja että se on valmis pelattavaksi. Tässä projektissa tason tiedetään olevan ladattuna heti, kun pelaajahahmo on saatavilla, joten sitä voidaan käyttää merkkinä tason latauksen ja tämän komennon loppumisesta. Unreal Engine 4:ssä on sisäänrakennettu tapa löytää pelaaja ja tarkistaa, että se on saatavilla. Esimerkkikoodissa 9 nähdään, miten komento odottaa pelaajahahmon löytymistä tasosta.

```
bool FLoadMapCommand::Update()
{
     ...
     APlayerController* PlayerController =
               GEngine->GetFirstLocalPlayerController(GWorld);

     if (IsValid(PlayerController))
     {
          APawn* PlayerCharacter = PlayerController->GetPawn();

          if (IsValid(PlayerCharacter))
          {
               /// Taso on ladattu
               return true;
          }
     }

     /// Taso ei ole vielä ladattu
     return false;
}
```

Esimerkkikoodi 9. Komennon logiikkaa, joka etsii pelaajaa ja odottaa, kunnes pelaaja on saatavilla. Käytetyt funktiot ovat sisäänrakennettuina UE4:ssä.

Tämä komento lisätään jonoon suorituskykytestien ajajassa, ja se ottaa parametrina ladattavan kartan (päätason) nimen merkkijonona. Esimerkkikoodissa 9 nähdään, miten komento lisätään testijuoksuun.

```
bool FPerformanceTestRunner::RunTest(const FString& InParameter)
{
      …
      FString MapPath = “MyMap”;
      ADD_LATENT_AUTOMATION_COMMAND(FLoadMapCommand(MapPath))
      …
}
```

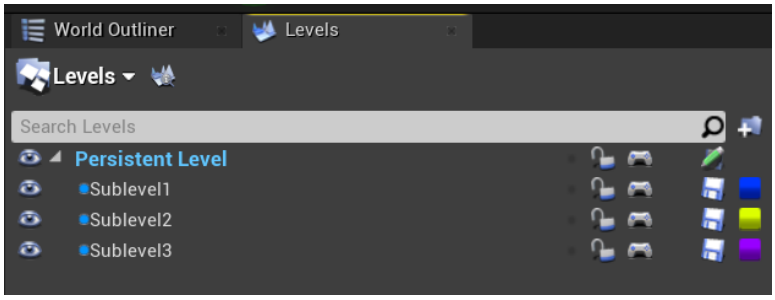
Esimerkkikoodi 9. FLoadMapCommand-komento lisätään testijuoksun komentojonoon. Parametri, joka sille annetaan, voi olla mikä vain merkkijono, kunhan sitä vastaava taso löytyy projektista.

Tason nimi voidaan lukea myös testijuoksun saamasta parametrista nimeltä InParameter. Tällä tavoin tätä testiluokkaa voidaan käyttää dynaamisesti useampaa eri tasoa varten.

4.2 Komento 2: Alitasojen suoratoisto

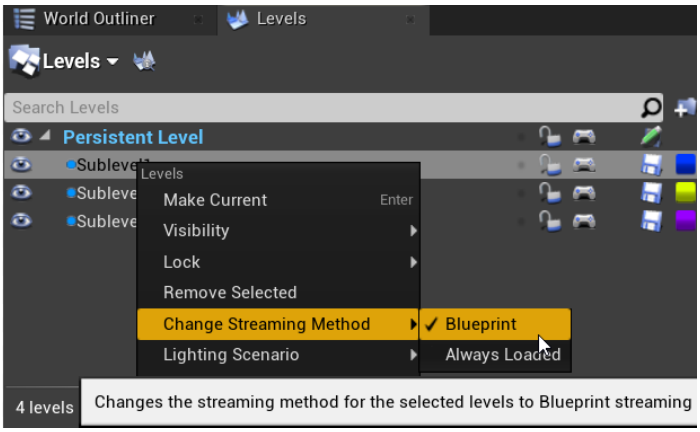
Unreal Engine 4:ssä tasot voidaan rakentaa alitasoista (engl. sublevels). Tämä tasojen paloittelu on melkeinpä välttämätöntä suurien peliprojektien kehityksessä. Kun tasot paloitellaan näin, jokaisella alitasolla on oma tiedostonsa ja ne toimivat itsenäisesti versionhallinnassa. Tällöin saman tason parissa voi työskennellä samanaikaisesti useampi kuin yksi henkilö, joista jokainen tekee muutoksia eri alitasoon.

Päätasossa voi olla useita alitasoja, ja alitaso voi olla käytössä monessa eri päätasossa samanaikaisesti. Päätasoa kutsutaan Unreal Engine 4:ssä pysyväksi tasoksi (engl. persistent level). Kuvassa 3 on esimerkki näkymästä, jossa päätasolle on luotu alitasoja.



Kuva 3. Alitasot näkyvät päätason Levels-näkymässä.

Alitasot eivät suoranaisesti ole osa päätasoa, mutta niiden sisältö suoratoistetaan sisään päätasoon. Alitasot voidaan suoratoistaa sisään päätasoon kahdella eri tavalla. Tämän asetuksen voi asettaa alitasokohtaisesti Levels-näkymässä, kuten kuvassa 4.



Kuva 4. Alitason suoratoistoasetuksen voi vaihtaa painamalla alitasoa päätason Levels-näkymässä oikealla hiiren näppäimellä. Tasojen vieressä oleva sininen piste ilmoittaa, että alitaso ladataan sisään Blueprint-metodilla.

Vaihtoehdon 1 (Blueprint) ollessa päällä alitasot eivät lataudu automaattisesti päätason mukana, vaan ne pitää ladata UE4:n blueprint-järjestelmän kautta visuaalisella ohjelmoinnilla. Yleensä tämä tehdään päätason omassa blueprint-logiikassa. Kun käytössä on vaihtoehto 2 (Always Loaded, suom. aina ladattu), alitaso ja kaikki sen sisältö ladataan aina automaattisesti päätasoon, kun päätaso ladataan.

Riippumatta suoratoistovaihtoehdosta alitasojen sisältö on aina nähtävissä päätason editorissa. Suorituskykytestejä varten luodut kamerapolut pidetään opinnäytetyön peliprojektissa omissa alitasoissaan, jotta ne voidaan ladata sisään hallitummin. Jokaisella kentällä on oma alitasonsa, joka sisältää kentän omat kamerapolut.

Kamerapolkujen alitaso voidaan lisäksi tarvittaessa pitää kokonaan erillään päätasosta ja ladata sisään vain testeissä koodin kautta, kuten tällä yksinkertaisella komennolla tehdäänkin. Komento oli opinnäytetyön alkaessa jo toteutettu.

Esimerkkikoodissa 10 nähdään, miten alitasot ladataan Unreal Engine 4:n sisäänrakennetun LoadLevelInstance-funktion avulla.

```
bool FStreamSublevelCommand::Update()
{
     /// Aloitetaan lataus kerran
     if (!bIsLoading)
     {
          /// Lataus alkaa
          bIsLoading = true;
          ULevelStreamingDynamic::LoadLevelInstance(
               GWorld, m_SublevelPath, FVector::ZeroVector,
               FRotator::ZeroRotator, bIsSuccess);
     }

     /// Palautetaan, kunnes lataus on onnistunut
     return bIsSuccess;
}
```

Esimerkkikoodi 10. Komennon logiikkaa, joka lataa alitason ja odottaa, kunnes se on latautunut sisään päätasoon. Ladattava alitaso valitaan FStreamSublevelCommand-komennolle syötetyn nimen perusteella.

Komento lisätään jonoon suorituskykytestien ajajassa tason latauksen jälkeen, ja se ottaa parametrina ladattavan alitason nimen merkkijonona. Esimerkkikoodissa 11 nähdään, miten komento lisätään testijuoksuun.

```
bool FPerformanceTestRunner::RunTest(const FString& InParameter)
{
     …
     FString SublevelPath = “MySublevel”;
     ADD_LATENT_AUTOMATION_COMMAND(
          FStreamSublevelCommand(SublevelPath))
     …
}
```

Esimerkkikoodi 11. FStreamSublevelCommand-komento lisätään testijuoksun komentojonoon. Parametri, joka sille annetaan, voi olla mikä vain merkkijono, kunhan sitä vastaava taso löytyy projektista.

4.3 Komento 3: Kamera-ajo

Kamera-ajon vaihe on suorituskykytestin tärkein vaihe, jossa itse kamera-ajo ja sen suorituskykydatan keräys eli profilointi tehdään. Siinä ajetaan

suorituskykytestejä varten luotu kamera yhden tai useamman kamerapolun läpi ja kerätään samalla dataa profiloijan avulla.

Tämäkin komento oli jo valmiiksi toteutettu opinnäytetyön alkaessa. Työn edetessä tähän komentoon tehtiin laajoja muutoksia, jotka käydään läpi luvuissa 4.3.1−4.3.5.

Ennen kuin komento voi aloittaa kamerapolkujen läpiajon, se alustaa itsensä funktiolla, joka sisältää kaiken kamera-ajoon liittyvän alustuslogiikan. Esimerkkikoodissa 12 nähdään, miten alustus aloitetaan komennon alussa.

```
bool FRunCameraSplinesCommand::Update()
{
     if (!bIsInitialized)
     {
          Initialize();
          return false;
     }
}
```

Esimerkkikoodi 12. FRunCameraSplinesCommand-komennon alussa kutsutaan luokan omaa Initialize-funktiota. Alustuslogiikka on jaettu omaan funktioonsa, sillä se sisältää paljon koodia, joka muuten tekisi Update-funktiosta vaikealukuisen.

Kamera-ajon alustus sisältää useamman eri vaiheen, joista tärkeimmät käydään läpi tarkemmin luvuissa 4.3.1−4.3.4. Lopuksi luvussa 4.3.5 nähdään, miten kamerapolkujen läpiajo tehdään.

4.3.1 Kamerapolkujen etsintä

Ensimmäisessä alustuksen vaiheessa (kamerapolkujen etsintä) etsitään testattavasta tasosta löytyvät kamerapolut ja järjestetään ne esiasetettuun järjestykseen. Poluissa on tätä varten editorissa järjestysnumero numeromuuttujan muodossa.

Alun perin tämä komento toimi vain yhdellä kamerapolulla, mutta jatkokehityksen aikana todettiin, että yhden suuren kamerapolun pilkkominen useampaan

pienempään kamerapolkuun auttaisi testejä olemaan tehokkaampia ajankäytössään. Useamman kamerapolun avulla on mahdollista testata tasosta useampi eri alue ilman, että kameran tarvitsee ensin kulkea alueelta toiselle. Sen sijaan kamera hyppää polulta toiselle aina, kun polku on käyty läpi.

Alustuksessa etsitään ja lisätään kamerapolut listaan, minkä jälkeen ne järjestetään järjestysnumeroiden perusteella. Lisäksi kamera-ajon ensimmäinen kamerapolku asetetaan olemaan ensimmäinen järjestyksestä löytyvä polku. Esimerkkikoodissa 13 nähdään, miten kamerapolut etsitään, järjestellään ja alustetaan.

```
bool FRunCameraSplinesCommand::Initialize()
{
     if (CameraSplineActors.Num() == 0)
     {
          /// Etsitään tasosta löytyvät kamerapolut
          UGameplayStatics::GetAllActorsOfClass(GWorld,
               ACameraSpline::StaticClass(), CameraSplineActors);

          for (const AActor Actor : CameraSplineActors)
          {
               if (ACameraSpline Spline = Cast<ACameraSpline>(Actor))
               {
                    CameraSplines.Add(Spline);
               }
          }

          /// Järjestetään kamerapolut järjestysnumeroiden perusteella
          CameraSplines.Sort([](const ACameraSpline& A,
               const ACameraSpline& B)
          {
               return A.OrderNumber < B.OrderNumber;
          }
     }

     /// Asetetaan kamera-ajon ensimmäinen kamerapolku
     if (!IsValid(CurrentCameraSpline))
     {
          CurrentCameraSpline = CameraSplines[0];
          CurrentCameraSplineIndex = 0;
     }
     …
}
```

Esimerkkikoodi 13. Initialize-funktio etsii tasosta löytyvät kamerapolut, järjestää ne järjestysnumeroiden perusteella ja asettaa kamera-ajon ensimmäisen kamerapolun.

4.3.2 Suorituskykykameran alustus

Toisessa alustuksen vaiheessa (suorituskykykameran alustus) odotetaan, että kaikki tason omat kamerat on alustettu, minkä jälkeen testin suorituskykykamera alustetaan ja otetaan pelaajan haltuun. Tällä tavoin vältytään ongelmilta, joissa hallittu kamera saattaa vahingossa siirtyä pois suorituskykykamerasta esimerkiksi pelaajan kameraan. Esimerkkikoodissa 14 nähdään, miten suorituskykykameran alustus aloitetaan.

```
void FRunCameraSplinesCommand::Initialize()
{
     …
     if (!bIsCameraPossessed)
     {
          /// Etsitään pelaajan kamera
          APlayerController* PlayerController =
               GEngine->GetFirstLocalPlayerController(GWorld);

          /// Tarkistetaan onko pelaajan kamera käyttökunnossa
          if (IsValid(PlayerController))
          {
               const APawn* PlayerCharacter =
                    PlayerController->GetPawn();

               if (!IsValid(PlayerCharacter))
               {
                    return;
               }
          }

          /// Alustetaan suorituskykytestin kamera
          InitializePerformanceCamera(PlayerController);
     }
     …
}
```

Esimerkkikoodi 14. Initialize-funktio aloittaa suorituskykykameran alustuksen, kun pelaajan kamera on löytynyt ja saatavilla. Suorituskykykameran alustuslogiikka on eristetty omaan funktioonsa, sillä se sisältää paljon koodia, joka muuten tekisi Initialize-funktiosta vaikealukuisen.

Alustettaessa suorituskykytestin kameraa komennon täytyy luoda uusi kameraohjain ja aktivoida se. Tämän jälkeen pelaajaohjaimelle voidaan antaa tämä uusi kameraohjain ja kameraohjaimelle voidaan luoda fyysinen olemus pelissä. Esimerkkikoodissa 15 nähdään, miten suorituskykytestin kameraohjaimen luonti ja haltuun ottaminen tehdään InitializePerformanceCamera-funktiossa.

```
void FRunCameraSplinesCommand::InitializePerformanceCamera(
     APlayerController* PlayerController)
{
     /// Luodaan suorituskykytestin kameraohjain
     FActorSpawnParameters SpawnInfo;
     SpawnInfo.Instigator = PlayerController->GetInstigator();
     PerformanceCameraController =
          GWorld->SpawnActor<APerformanceCameraController>(
               APerformanceCameraController::StaticClass(), SpawnInfo);

     /// Aktivoidaan kameraohjain pelaajaohjaimen tiedoilla
     PerformanceCameraController->OnActivate(PlayerController);

     /// Vaihdetaan pelaajan ohjain suorituskykytestin kameraohjaimeksi
     PlayerController->Player->SwitchController(
          PerformanceCameraController);

     /// Luodaan kameralle Actor
     CameraActor = GWorld->SpawnActor(ADefaultPawn::StaticClass());

     // Luodaan kameralle Pawn
     DefaultCameraPawn = Cast<ADefaultPawn>(CameraActor));
     DefaultCameraPawn->SetActorTickEnabled(false);
     DefaultCameraPawn->SetActorEnableCollision(false);
     DefaultCameraPawn->SetRole(ROLE_Authority);

     // Annetaan Pawn suorituskykytestin kameraohjaimen haltuun
     PerformanceCameraController->Possess(DefaultCameraPawn);

     bIsCameraPossessed = true;
}
```

Esimerkkikoodi 15. InitializePerformanceCamera-funktio luo suorituskykytestin kameraohjaimen ja aktivoi sen. Uusi kameraohjain otetaan käyttöön pelaajakameran tilalle, ja sille luodaan fyysinen olemus pelissä.

Kun suorituskykytestin kameraohjain aktivoidaan, se ottaa sisäänsä pelaajaohjaimen ja alustaa itsensä sen tiedoilla. Liitteessä 4 on esimerkki APerformanceCameraController.h-luokasta.

4.3.3 Käyttöliittymän piilotus

Kolmannessa alustuksen vaiheessa (käyttöliittymän piilotus) piilotetaan kaikki pelin käyttöliittymät, jotta ne eivät vaikuttaisi suorituskykytestin tuloksiin. Mikäli tätä vaihetta ei tehdä, käyttöliittymä saattaa peittää asioita ruudulta tai tuoda ylimääräistä rasitusta muistiin, mikä tekee testituloksista epäluotettavia. Esimerkkikoodissa 17 on ote komennon alustuslogiikan koodista, jossa käyttöliittymän piilotus aloitetaan.

```
void FRunCameraSplinesCommand::Initialize()
{
     …
     if (!bIsUIHidden)
     {
          /// Piilotetaan UI
          APlayerController* PlayerController =
               GEngine->GetFirstLocalPlayerController(GWorld);
          HideUI(PlayerController);
     }
     …
}
```

Esimerkkikoodi 17. Initialize-funktio aloittaa käyttöliittymän piilotuksen. Logiikka on eristetty omaan funktioonsa, sillä se sisältää paljon koodia, joka muuten tekisi Initialize-funktiosta vaikealukuisen.

Tämä vaihe etsii kaikki tietyn tyypin Widget-oliot, ja piilottaa ne. Widget on UE4-pelimoottorin käyttöliittymäkomponentti, josta kaikki pelin käyttöliittymän elementit perivät.

Kehityksen aikana törmättiin ongelmaan, jossa käyttöliittymä ei joskus piiloutunut oikein. Tähän löytyi ratkaisuksi PlayerController-luokan ClientSetCinematicMode_Implementation-funktio, joka asettaa pelaajaohjaimen elokuvatilaan, jotta mikään ei varmasti vaikuta kameran liikkeeseen tai käyttöliittymään. Esimerkkikoodissa 18 nähdään yksi mahdollinen ratkaisu, jolla käyttöliittymä voidaan piilottaa.

```
void FRunCameraSplinesCommand::HideUI(
     APlayerController* PlayerController)
{
     /// Etsitään kaikki UUserWidget-luokan oliot
     TArray<UUserWidget*> FoundWidgetsList;
     UWidgetBlueprintLibrary::GetAllWidgetsOfClass(Gworld,
          FoundWidgetsList, UUserWidget::StaticClass());

     /// Iteroidaan löydettyjen UUserWidget-olioiden läpi
     /// ja piilotetaan ne
     for (auto Iterator = FoundWidgetsList.CreateConstIterator();
          Iterator; Iterator++)
     {
          if (UUserWidget* Widget = Cast<UUserWidget>(*Iterator);
               IsValid(Widget))
          {
               Widget->RemoveFromParent();
               bIsUIHidden = true;
          }
     }

     /// Asetetaan päälle elokuvatila (Cinematic Mode)
     PlayerController->ClientSetCinematicMode_Implementation(true,
          true, true, true);
}
```

Esimerkkikoodi 18. HideUI-funktio etsii kaikki UUserWidget-luokan oliot ja piilottaa ne. Lisäksi se asettaa pelaajaohjaimelle päälle elokuvatilan, jotta se ei vaikuta käyttöliittymään.

4.3.4 Profiloijan alustus

Suorituskykytestin alustuksen lopuksi alustetaan ja käynnistetään profiloija. Se kerää dataa ruudunpäivityksestä ja muistin käytöstä joka ruudunpäivityksellä. Esimerkkikoodissa 19 nähdään, miten profilointi käynnistetään.

```
bool FRunCameraSplinesCommand::Initialize()
{
     …
     if (!bIsProfilerRunning)
     {
          /// Etsitään profiloija ja aloitetaan profilointi
          FCsvProfiler* Profiler = FCsvProfiler::Get();
          Profiler->BeginCapture();

          bIsProfilerRunning = true;
     }
     …
}
```

Esimerkkikoodi 19. Initialize-funktio käynnistää FCsvProfiler-luokan profiloijan.

Lisäksi alustetaan delegaattikahva, joka on kiinni UE4:n sisäänrakennetussa funktiossa, joka kutsuu sitä kuuntelevia delegaattifunktioita aina, kun peli jäätyy. Esimerkkikoodissa 20 nähdään, miten testin oma jäätymisiin reagoiva delegaattifunktio kiinnitetään UE4:n sisäänrakennettuun OnHitchDetectedDelegate-delegaattikahvaan.

```
bool FRunCameraSplinesCommand::Initialize()
{
     …
     if (!bIsHitchDetectionInitialized)
     {
          HitchDetectionDelegateHandle =
               GEngine->OnHitchDetectedDelegate.AddRaw(this,
               &FRunCameraSplinesCommand::HandleEngineHitchDetection);
          bIsHitchDetectionInitialized = true;
     }
     …
}
```

Esimerkkikoodi 20. Initialize-funktio liittää testin oman HandleEngineHitchDetection-funktiokutsun UE4:n sisäänrakennettuun OnHitchDetectedDelegate-delegaattiin ja alustaa sillä HitchDetectionDelegateHandle-delegaattikahvan.

Kun jäätyminen havaitaan, siitä halutaan ottaa kuvakaappaus. Poikkeuksena ei kuitenkaan oteta huomioon jäätymisiä, joissa pelin säie prosessorissa ei jäätynyt. Tällä tavoin saadaan tietoa vain jäätymisistä, jotka vaikuttivat peliin ja jotka pelaaja voi nähdä. Esimerkkikoodissa 21 nähdään, miten jäätymisistä tallennetaan kuva laitteelle.

```
void FRunCameraSplinesCommand::HandleEngineHitchDetection(
     const EFrameHitchType HitchType, float HitchDurationInSeconds)
{
     if (HitchType != EFrameHitchType::GameThread)
     {
          /// Pelin säie ei jäätynyt.
          return;
     }

     /// Etsitään profiloija ja otetaan talteen tason nimi,
     /// ruudun numero ja kameran sijainti
     FCsvProfiler* Profiler = FcsvProfiler::Get();
     const Fstring WorldName = GWorld->GetMapName(true);
     const int32 FrameNumber = Profiler->GetCaptureFrameNumber();

     FVector CameraPosition = CameraPawn->GetActorLocation();

     /// Otetaan kuvakaappaus
     const FString PictureName = FString::Printf(
          TEXT("Hitch_%s_%d_(%s)"),
          *WorldName, FrameNumber, *CameraPosition.ToString());
     FScreenshotRequest::RequestScreenshot(PictureName, false, false);
}
```

Esimerkkikoodi 21. HandleEngineHitchDetection-funktio ottaa kuvakaappauksen ja tallentaa sen. Kuvakaappaus nimetään mallilla, josta käy ilmi tason nimi, ruutunumero ja kameran sijainti kuvan hetkellä.

Tämä vaihe komennossa toimi moitteettomasti, joten siihen ei ole tarvinnut tehdä muutoksia.

4.3.5 Kamerapolkujen läpiajo

Kamerapolkujen läpiajo on hyvin yksinkertainen prosessi. Ajon aikana kameran sijaintia ja suuntaa päivitetään pitkin nykyistä kamerapolkua. Aina kun kamerapolku loppuu, siirrytään seuraavaan, kunnes kaikki kamerapolut on käyty läpi. Lopuksi testin profiloijan profilointi lopetetaan, jolloin kerätty profilointidata tallennetaan laitteelle. Esimerkkikoodissa 22 nähdään, miten kamera ajetaan kaikkien kamerapolkujen läpi.

```
bool FRunCameraSplinesCommand::Update()
{
     …
     /// Päivitetään kameran sijainti polun mukaisesti
     const FVector CurrentLocation =
          CameraSpline->GetCurrentPositionAtTime();
     DefaultCameraPawn->SetActorLocation(CurrentLocation);

     /// Päivitetään kameran kiertosuunta polun mukaisesti
     const FRotator CurrentRotation =
          CameraSpline-GetCurrentRotationAtTime();
     DefaultCameraPawn->SetActorRotation(CurrentRotation);

     /// Jatketaan polkua eteenpäin
     CameraSpline->Tick(GWorld->DeltaTimeSeconds);


     if (CurrentCameraSpline->HasTimeEnded())
     {
          if (++CurrentCameraSplineIndex < CameraSplines.Num())
          {
               /// Vaihdetaan seuraavaan polkuun
               CurrentCameraSpline =
                    CameraSplines[CurrentCameraSplineIndex++];
          }
          else
          {
               /// Lopetetaan pelin jäätymisen seuranta
               GEngine->OnHitchDetectedDelegate.Remove(
                    HitchDetectionDelegateHandle);

               /// Lopetetaan profilointi
               FCsvProfiler::Get()->EndCapture();

               /// Komento loppui. Lopetetaan.
               return true;
          }
     }

     /// Komento ei ole vielä loppunut. Jatketaan.
     return false;
}
```

Esimerkkikoodi 22. Update-funktio päivittää suorituskykykameran sijaintia polkua pitkin ja kutsuu kamerapolun Tick-funktiota. Kun polku loppuu, vaihdetaan seuraavaan, kunnes kaikki polut on käyty läpi.

Työn aikana huomattiin, että useamman kamerapolun ollessa käytössä testiajon aikana olisi hyödyllistä nähdä, millä kamerapolulla kamera on milläkin hetkellä. Tätä varten komentoon lisättiin jatkuvasti ruudulle päivittyvä viesti, kuten esimerkkikoodissa 23 nähdään.

```
bool FRunCameraSplinesCommand::Update()
{
     …
     GEngine->AddOnScreenDebugMessage(-1,
          GWorld->GetDeltaSeconds(),
          FColor::Green, FString::Printf(TEXT("Camera path %i/%i"),
          CurrentCameraSplineIndex+1, CameraSplines.Num()));
     …
}
```

Esimerkkikoodi 23. Update-funktio päivittää ruudulle viestin, joka näyttää senhetkisen aktiivisen kamerapolun järjestysnumeron ja kamerapolkujen kokonaismäärän.

Tästä kaikesta koostuva lopullinen suorituskykytestin ajava komento lisätään jonoon suorituskykyjen ajajassa tason latauksen ja alitasojen suoratoiston jälkeen. Lopuksi tämäkin komento lisätään testiajoon jonoon muiden komentojen perään, kuten esimerkkikoodi 24 näyttää.

```
bool FPerformanceTestRunner::RunTest(const FString& InParameter)
{
     …
     ADD_LATENT_AUTOMATION_COMMAND(FRunCameraSplinesCommand())
     …
}
```

Esimerkkikoodi 24. FRunCameraSplinesCommand-komento lisätään testijuoksun komentojonoon.

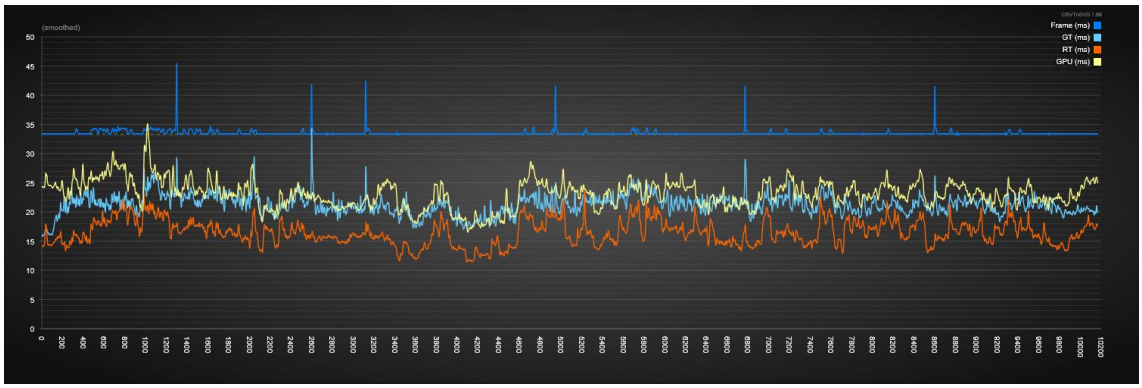
5 Testitulosten keräys ja visualisointi

Testiajon jälkeen kerätään testitulokset. Tämä tehdään lukemalla ja formatoimalla CSV-tiedosto, jonka FCsvProfiler-luokka luo testin lopuksi. Tämä taulukkomuotoinen tiedosto löytyy projektin `/Saved/Profiling/CSV`-kansiosta. Se käännetään lopuksi esimerkkikoodin 25 mukaisella komennolla luettavampaan vektorikuvalliseen SVG-tiedoston muotoon Unreal Engine 4:n CSVToSVG-työkalua käyttäen.

```
CSVToSVG.exe -csvdir "<MyProject\Unreal\MyGame\Saved\Profiling\CSV>" -
o "<path/to/output/PerformanceTestGraph.svg>" -stats "Frame (ms)" "GT
(ms)" "RT (ms)" "GPU (ms)"
```

Esimerkkikoodi 25. Ote komennosta, joka käyttää CSVToSVG-työkalua luomaan CSV-tiedostosta SVG-tiedoston.

Esimerkkikoodin 25 komennon voi ajaa manuaalisesti Windows-tietokoneen komentorivillä. Lopputuloksena saadaan kuvan 5 kaltainen graafi, josta voidaan nähdä kerätty data käyrien muodossa.

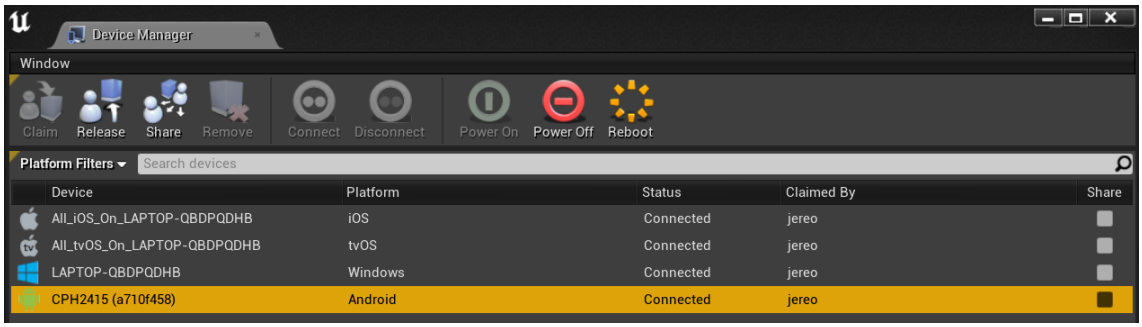


Kuva 5. Graafi visualisoi käyrillä profiloijan keräämää dataa [11, kuva 1].

6 Testien ajo Android-puhelimella

Tässä luvussa käydään läpi tapaustutkimuksen pohjana olevan peliprojektin tapa ajaa automaattisia suorituskykytestejä puhelimella UE4-editorin sijaan. Mobiilipelejä testattaessa testien ajo suoraan fyysisellä mobiililaitteella on paras tapa saada mahdollisimman tarkkoja ja todenmukaisia tuloksia. Kirjoitushetkellä projektin automaattisten testien testausprosessi oli toteutettu ja todettu toimivaksi vain Android-puhelimilla.

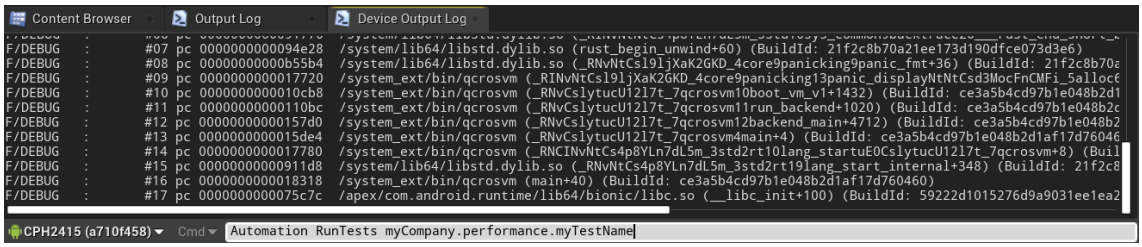
Testien ajo puhelimella vaatii, että Unreal Engine on linkitetty Android-laitteeseen. Linkitys tapahtuu automaattisesti, kunhan puhelin on piuhalla kiinni tietokoneessa ja UE4-pelimoottorin Android-asetukset on asetettu oikein. Linkityksen voi tarkistaa UE4:n Device Manager -näkymästä, kuten kuvassa 6 nähdään.



Kuva 6. Android-laite näkyy Device Manager -näkymässä yhdistyneenä.

Kun Unreal Engine on linkitetty puhelimeen ja puhelimelle on asennettu projektin pelisovellus, testejä voidaan ajaa suoraan puhelimella.

Peli avataan puhelimella ihan kuin tavallisestikin, eli painamalla sovelluksen kuvaketta. Kun sovellus on auki, avataan tietokoneella UE4:n näkymä nimeltä Device Output Log ja syötetään sinne komento, joka ajaa testin. Kuva 7 on esimerkki siitä, miltä komennon syöttö näyttää Device Output Log -näkymässä.



Kuva 7. Testin aloittava komento ollaan syöttämässä Device Output Log -näkymään, ja laitteeksi on valittu piuhalla liitetty Android-puhelin.

Kun suorituskykytesti on ajettu loppuun saakka, sen profiloija tallentaa testitulokset laitteelle automaattisesti. Testitulokset löytyvät puhelimelta kansiosta “/storage/emulated/0/UE4Game/MyGame/Saved/Profiling/CSV/”, jossa MyGame on projektin nimi. Profilointidata on CSV-tiedoston muodossa nimellä “Profile(YYYYMMDD_HHMMSS).csv” (esim. Profile(20230315_160249).csv).

Tällä prosessilla oli työn aikana mahdollista varmistaa, että uudet muutokset testeihin toimivat, ennen kuin muutokset puskettiin versionhallintaan. Windows-tietokoneen ja Unreal Enginen asettaminen valmiiksi Android-kehitykseen oli haastavaa, mutta saatiin loppujen lopuksi toimimaan, eikä prosessia tarvinnut muuttaa.

7 Testiajojen automatisointi

Automatisoitujen testien ajo manuaalisesti vie aikaa ja resursseja, vaikkakin vähemmän kuin täysin manuaalinen testaus. Tämäkin prosessi voidaan onneksi automatisoida ja säästää kehittäjien aikaa luovempaan työhön.

Jatkuva integraatio (engl. Continuous Integration, lyh. CI) on ohjelmistokehitysmenetelmä, jossa ohjelmistomuutosten tapahtuessa muutokset integroidaan automaattisesti projektin versionhallintaan, jolloin sovelluksesta rakennetaan uusi versio jokaiselle alustalle ja olemassa olevat automaattiset testit ajetaan uutta projektiversiota vasten.

Tätä kokonaisuutta kutsutaan automatisointiputkeksi. Putki mahdollistaa muutosten integraation ja sen seurauksena muodostuvan uuden ohjelmistokokonaisuuden analysoinnin, testauksen ja julkaisun. Tämä prosessi tapahtuu yleensä etänä toisella tietokoneella (rakennuskone), jolle annetaan tehtäviä pilvessä ja vapautetaan näin kehittäjä jatkamaan pelin kehitystä samalla, kun rakennuskone tekee tehtävänsä. Jos jokin rakennuskoneen tehtävistä epäonnistuu, kehittäjän täytyy korjata muutoksensa toimimaan integraatiossa, ennen kuin hän voi jatkaa tulevien muutoksien kehitystä.

Ilman kattavaa automatisointiputkea muutosten integrointi muuntuu usein harvinaiseksi ja umpimähkäiseksi, sillä kehittäjät välttävät testauksen ja integraation tekemistä manuaalisesti ja jäävät siksi helposti kehittämään muutoksiaan yksin omassa versionhallinnan haarassaan pidempään kuin tarpeellista. Näin ollen kehittäjien ja tuottajien tai asiakkaiden välinen palautesilmukka on pitkä ja satunnainen. Automatisointiputki auttaa kehittäjiä testaamaan muutoksensa helpommin ja integroimaan muutoksensa useammin ja näin ollen saamaan myös paljon enemmän palautetta. [12.]

Tämän opinnäytetyön peliprojektissa automatisointiputki käyttää TeamCityä, AWS-laitefarmia ja Appiumia. Nämä palvelut ovat tämän opinnäytetyön aiheen ulkopuolella, mutta ne selitetään seuraavaksi lyhyesti.

TeamCity on alusta automatisointiputkien hallinnalle [13], jolla tässä projektissa rakennetaan pelistä uusia versioita eri alustoille ja testataan niitä automatisoidusti. Se rakentaa ja testaa pelin uusimman version tasaisin väliajoin, jotta kehitystiimi voi saada tietoa isoista ongelmista heti, kun ne havaitaan versionhallinnassa.

AWS Device Farm on mobiililaitteita vuokraava pilvipalvelu [14], jota käytetään tässä projektissa ajamaan suorituskykytestejä isolla määrällä erilaisia mobiililaitteita. AWS Device Farm -palvelua käytetään suoraan TeamCityn automatisointiputkessa suorituskykytestien ajamista varten.

Appium on mobiililaitteille tarkoitettu avoimen lähdekoodin testausautomaatiokehys [15], jolla automatisoidaan tässä projektissa esimerkiksi mobiilisovelluksien asennus ja avaus laitteella sekä käyttöliittymän ohjaus erilaisten skriptien avulla. Appium-kehystä käytetään suoraan AWS Device Farm -palvelun kanssa ohjaamaan vuokrattuja mobiililaitteita.

8 Havaitut ongelmat ja parhaat käytännöt

Työn aikana törmättiin useampaan ongelmaan. Ongelmien ilmaantuessa lähestymistapa niihin oli kokeilullinen. Ensin tutkittiin ja listattiin kaikki vaihtoehdot, minkä jälkeen valittiin parhaalta kuulostava ratkaisuidea, ja lopuksi kokeiltiin, toimiiko ratkaisuidea käytännössä. Jos ei, siirryttiin toiseksi parhaaseen vaihtoehtoon.

Koko prosessin aikana pidettiin yllä jatkuvaa muistiinpanojen kirjoittamista erityisesti prosessin aikana opituista asioista. Joskus muistiinpanoista kehittyi uusia ratkaisuideoita, jotka osoittautuivat paljon paremmiksi kuin alkuperäiset ideat.

Joissain ongelmissa vaadittiin aluksi sen järjestelmän tutkimista, jossa ongelma piili, jotta voitiin saada syvempi ymmärrys ongelma-alueesta ja selvittää, mitä mahdollisia ratkaisuvaihtoja ongelmaan on.

8.1 Ongelma 1: Automaattiset välianimaatiot

Joidenkin kenttien latauksen yhteydessä törmättiin automaattisesti alkaviin välianimaatioihin. Nämä välianimaatiot peittivät koko ruudun, jolloin ne häiritsivät testien kamera-ajoja näkemästä tasoa, mikä johti siihen, että testit eivät tuottaneet hyödyllisiä testituloksia.

Näissä tapauksissa parasta on löytää keino joko asettaa välianimaatiot kokonaan pois käytöstä tai ohittaa ne heti, kun ne aktivoituvat. Tässä projektissa välianimaatiot ohitettiin, sillä se oli helpompaa kuin niiden poistaminen testeistä.

8.2 Ongelma 2: Käyttöliittymä

Pelissä on käyttöliittymä, joka peittää osan ruudusta ja jonka grafiikka vaikuttaa suoraan pelin suorituskykyyn. Näin ollen on testien edun mukaista piilottaa käyttöliittymä, jotta saavutettaisiin mahdollisimman tasokeskeiset testitulokset, riippumatta muutoksista käyttöliittymään.

Unreal Engine 4 tarjoaa tähän helpon ratkaisun APlayerController::ClientSetCinematicMode_Implementation()-funktion ja UWidgetBlueprintLibrary::GetAllWidgetsOfClass()-funktion avulla. Esimerkkikoodi 18 on luvussa 4.3.3 (Käyttöliittymän piilotus).

8.3 Ongelma 3: Testien pituus

Testien rakentamisessa on tärkeää pitää mielessä testiajon viemä aika eli testien yhteinen pituus. Hyvät testit ovat nopeita ajaa, eivät käytä tarvittua enempää aikaa ja tarjoavat vain tarvittavan määrän tietoa. Hyvät testit eivät myöskään toista samaa, mitä muut testit jo testaavat, jollei se ole välttämätöntä.

Mikäli testien pituudesta ei pidetä huolta, testien yhteinen pituus kasvaa räjähdysmäisesti. Ongelma kasvaa, kunnes testien ajo kestää niin kauan, että se ei enää ole käytännöllistä tai testejä on niin paljon, että tarvittavia testituloksia on vaikea löytää ja lukea. Tällöin testikokonaisuuden ajoa vältellään tai sitä tehdään harvemmin, jolloin menetetään kokonaan testien tarjoaman jatkuvan palautteen tuoma arvo.

Ratkaisu tähän ongelmaan on helppo. Jos huomataan, että testin ajoon käytetään paljon aikaa, sitä joko optimoidaan lyhyemmäksi tai se erotetaan muista testeistä, jotta sen voi ajaa erikseen. Suorituskykytestien kohdalla optimointi tarkoittaa yleensä kamerapolkujen lyhennystä tai nopeuttamista. Jos esimerkiksi kamerapolusta löytyy kohtia, joissa ei ole erityisen paljoa nähtävää, ne voidaan poistaa testistä kokonaan. Jos taas tasossa on paljon avoimia kohtia, joissa polku kulkee suoraan, voidaan kamerapolkua nopeuttaa hieman ilman, että se aiheuttaa liikaa rasitusta laitteelle.

Opinnäytetyön peliprojektissa testien pituus on edelleen ongelma, vaikka testejä optimoitiin hieman. Tämä johtuu suurimmalta osin siitä, että uusien testien rakennus on ollut tärkeämpää ja testien pituus ei ole ollut vielä tarpeeksi iso ongelma, että se pitäisi ratkaista välittömästi. Olen kuitenkin varma, että testejä

optimoidaan lähitulevaisuudessa, ennen kuin optimointioperaatiosta tulee liian suuri.

8.4 Ongelma 4: Pelaajan aktivoimat tapahtumat

Työn aikana joissain tasoissa törmättiin ongelmiin sellaisten tapahtumien kanssa, jotka eivät ole aktivoituna oletusarvoisesti vaan jotka vain pelaaja voi aktivoida. Aktivointi manuaalisesti tehdään yleensä pelaajan toimesta joko lähestymällä tapahtuma-aluetta tai painamalla jotain näppäintä tapahtumaolion lähellä. Näitä ovat esimerkiksi vivut ja napit sekä alueet, joilla viholliset hyppäävät pelaajan päälle tyhjästä. Tapahtumatyyppejä on kahdenlaisia: pysyviä ja hetkellisiä.

Pysyvät tapahtumat ovat tapahtumia, jotka aktivoituessaan pysyvät päällä eivätkä koskaan lopu. Esimerkki pysyvästä tapahtumasta on vihollisten ilmestyminen maan alta, jossa tapahtuman aktivoituessa viholliset ilmestyvät eivätkä koskaan katoa. Pysyvien tapahtumien aktivointi ja testaaminen on helppoa. Nämä tapahtumat voidaan aktivoida testin alussa, jolloin ne ovat edelleen aktivoituna ja testattavissa, kun testin kamera saapuu niiden kohdalle.

Hetkelliset tapahtumat ovat tapahtumia, jotka tekevät aktivoituessaan jonkinlaisen hetkellisen muutoksen tasossa, kuten hissin liikuttamisen ylös ja alas. Hetkellisten tapahtumien aktivointi ja testaaminen on hieman haastavampaa, sillä ne täytyy aktivoida vasta, kun kamera on tarpeeksi lähellä ja näkee tapahtuma-alueen. Muuten kamera ei näe tapahtumaa ja tapahtumasta ei saada dataa suorituskykytesteissä. Nämä tapahtumat on paras asettaa aktivoitumaan erillisellä testiä varten luodulla aktivointialueella, joka aktivoi sen vastuulle asetetun tapahtuman heti, kun kamera osuu aktivointialueen kohdalle.

Tässä peliprojektissa tuli vastaan muutama tapaus, joissa pysyviä tapahtumia piti aktivoida. Päätös aktivointiin tehtiin, koska haluttiin, että kentästä löytyy kaikki aktivoitavat tapahtumat, jotta päästiin mahdollisimman lähelle todellista pelitilannetta.

8.5 Ongelma 5: Satunnaisesti generoidut tasot ja elementit

Testin eri ajojen välillä tason ja sen sisältä löytyvien elementtien täytyy olla yhtenäisiä, jotta testituloksia voidaan vertailla. Muussa tapauksessa testituloksissa voi olla joka ajon välillä eroja, jotka eivät johdu välttämättä siitä, että pelin suorituskyky on laskenut. Näin ollen testituloksia on vaikea käyttää hyödyksi tarkkana tietona pelin suorituskyvystä. Testit on siis hyvä rakentaa yhtenäisyys mielessä, jotta testituloksia voidaan käyttää kaikista tehokkaimmin hyödyksi.

Työn aikana kohdattiin ongelma tämänkaltaisessa satunnaisesti generoitujen tasojen ja niiden elementtien testauksessa. Useimmiten peleissä näitä tasoja ei ole olemassa, ennen kuin peli on avattu, taso avattu ja taso generoitu. Lisäksi tasot generoidaan aina satunnaisesti, jollei niitä ole konfiguroitu generoitumaan aina samalla tavalla.

Työn aikana kohdattiin esimerkiksi tasoja, jotka generoidaan joka peluukerralla uudelleen. Tähän sisältyy kaikki tason geometria ja sen sisältö. Tasot koostuvat tiilistä, joita on projektissa satoja erilaisia. Jokaisella tasolla on oma kokoelmansa tiiliä, joita se käyttää tason generoinnissa. Jokaisessa tiilessä on geometriaa, esiasetettuja paikkoja, joihin voidaan generoida sisältöä, sekä muutama uloskäynti, johon muut tiilet kiinnittyvät. Kun taso avataan, pelin tasogeneraattori valitsee projektista satunnaisesti useamman tiilen ja rakentaa niistä pelattavan tason. Tämän jälkeen tason sisälle generoidaan sisältöä, kuten vihollisia, satunnaista keräiltävää sekä erilaisia uniikkeja tapahtumia, jotka pelaaja voi aktivoida.

Tästä tasorakenteesta seuraa useampi ongelma, joihin keksittiin kaikkiin työn aikana omat ratkaisunsa.

Ongelma: Kaikkia tason eri mutaatioita on mahdoton testata järkevässä ajassa.
Ratkaisu: Testiä varten valittiin pahin mahdollinen kokoonpano tason generaatiosta ja käytettiin sitä testaamaan kentän alin mahdollinen suorituskyky.

Ongelma: Kaikkia tason mahdollisia tiiliä ei voi sisällyttää samaan testiin ilman, että se vaikuttaa suorituskykyyn huomattavasti.
Ratkaisu: Testiä varten valittiin pahimmat mahdolliset tiilet, mutta rajoitettiin generoitujen tiilien määrä kuitenkin lähelle todellista pelitilannetta.

Ongelma: Testitason generointi pitää konfiguroida yhtenäiseksi.
Ratkaisu: Tason ja sen sisällön generointiin käytettyihin satunnaislukulähteisiin esivalittu siemenarvo asetettiin etukäteen, jolloin generoinnin lopputulos pysyi aina yhtenäisenä testien välillä.

Ongelma: Kamerapolkujen rakennus on vaikeaa ilman editorissa näkyvää tasoa, jonka päälle polut rakentaa.
Ratkaisu: Lopullisesta tasogeneraation luomasta testitasosta rakennettiin väliaikainen kopio editorissa, jonka päälle alitaso ja sen kamerapolut voitiin rakentaa.

8.6 Ongelma 6: Laitefragmentaatio

Laitefragmentaatio (engl. device fragmentation) tarkoittaa ongelmia, jotka ilmenevät vain joillain tietyillä laitteilla niiden ominaispiirteitten takia. Nämä ongelmat voivat johtua esimerkiksi erikokoisista näytön resoluutioista tai siitä, että laitteet käyttävät vanhentunutta käyttöjärjestelmää. Ongelma on yleisempi Android-puhelimilla, sillä niitä valmistavat eri yritykset ympäri maailmaa eri tavoin. Applen iOS-laitteilla ongelmat ovat harvinaisempia, mutta niitä ilmenee kyllä Applen vanhemmilla iOS-laitteilla. [16; 17.]

Jotta tästä johtuvilta ongelmilta voidaan välttyä, tarvitaan tarkat määritelmät siitä, mitkä laitteet ja käyttöjärjestelmät ovat tuettuja. Lisäksi pelin virallisesti tuetut laitteet tulee testata läpikotaisin osana automatisointiputkea, jotta ongelmat eri laitteilla voidaan havaita ja korjata, ennen kuin peli pääsee pelaajien pelattavaksi.

Opinnäytetyön projektissa peliä ei esimerkiksi oltu testattu kirjoituksen hetkellä ollenkaan automaattisesti iOS-laitteilla. Ainoat testit iOS-laitteilla oli tehty

manuaalisesti. Automaattinen testausprosessi iOS-laitteille on kuitenkin aktiivisesti kehitteillä.

Lisäksi välillä jotkut kentät eivät lataudu laisinkaan vanhemmilla puhelimilla muistin puutteen vuoksi, jolloin kenttien muistin käyttöä optimoidaan tarpeen mukaan.

8.7 Ongelma 7: Verkko-ongelmat

Verkossa pelattavien pelien suorituskyky kärsii joskus verkko-olosuhteitten mukaan. Tämä johtuu usein liian pienestä kaistanleveydestä, suuresta viiveestä ja/tai pakettihävikistä. [17.]

Verkko-ongelmat johtavat esimerkiksi siihen, että pelin ja pelipalvelimen kommunikaatio ei toimi, mikä näkyy pelaajalle viiveinä ja virheinä pelimaailmassa ja käyttöliittymässä.

Verkko-ongelmilta ei voi välttyä, mutta niitä voidaan lievittää. Avain tähän on jatkuva suorituskykytestaus eri verkko-olosuhteissa, jotta ongelmia voidaan havaita ja lievittää ajoissa. Yksi tapa lievittää esimerkiksi kaistanleveyden ja pakettihävikin ongelmia on paloitella lataukset useaan pieneen pakettiin, jolloin yksittäisen paketin lataus on nopeaa ja pakettihävikin vaikutus on minimaalista.

9 Yhteenveto

Opinnäytetyön tavoitteena oli tutkia, miten työprojektissa tehdään suorituskykytestausta Unreal Engine 4 -pelimoottorilla. Lisäksi rakennettiin lisää testejä ja paranneltiin olemassa olevia työkaluja ja automaattisia prosesseja. Lopputuloksena saatiin hyvä ymmärrys kokonaisuudesta ja kirjoitettiin tämä opinnäytetyöraportti kuvaamaan koko prosessi testien rakennuksesta lähtien testien ajoon Android-puhelimella.

Työn aikana opittiin, että suorituskykytestauksen automaatio koostuu monesta eri vaiheesta ja palvelusta. Ensin rakennetaan testit valitussa pelimoottorissa ja sen testausautomaatiokehyksessä, minkä jälkeen testit juoksutetaan automaattisesti valitulla automatisointiputkella laitefarmia hyödyksi käyttäen. Prosessi on valtava, ja jokaisessa vaiheessa on omat erilaiset haasteensa, joita kaikkia ei ole mahdollista käydä läpi yhdessä opinnäytetyössä.

Lisäksi saatiin syvempi ymmärrys Unreal Engine -pelimoottoriin ja Unreal Automation System -kehykseen. Unreal Engine on oiva vaihtoehto pelien automaattiseen testaamiseen, sillä se tarjoaa runsaasti omia sisäänrakennettuja työkaluja, joilla automaatiotestaus tapahtuu erittäin luontevasti ja vaivatta. Esimerkiksi kamerajuoksut voi toteuttaa UE4:n sisäänrakennettua SplineComponent-luokkaa laajentamalla, ja suorituskyvyn profilointikin onnistuu helposti CsvProfiler-luokan avulla.

Insinöörityössä kehitettiin olemassa olevia suorituskykytestien luontiin ja ylläpitoon liittyviä työkaluja lisäämällä esimerkiksi useampien kamerapolkujen tuki ja staattisien kameroiden toiminnallisuus. Lisäksi suunniteltiin tapa testata satunnaisesti generoituja kenttiä ja luotiin paljon uusia automaattisia suorituskykytestejä.

Työn tuloksista kävi ilmi, että pelien automaattinen suorituskykytestaus on yllättävän vaikeaa. Esimerkiksi pelkkä suorituskykytestien rakennus sisältää paljon omia haasteitaan. Testien pituus tulee nopeasti ongelmaksi, automaattiset välianimaatiot ja käyttöliittymä täytyy ohittaa ja piilottaa, ja pelaajan aktivoimat

tapahtumat täytyy ottaa huomioon. Lisäksi satunnaisesti generoidun sisällön testaaminen vaatii paljon suunnittelua ja pragmaattista päätöksentekoa, laitefragmentaatio vaatii laajaa testausta eri laitteilla, ja peliä täytyy myös testata erilaisissa verkko-olosuhteissa, jotta saadaan tietoa mahdollisista verkko-ongelmista. Kaikki tämä on otettava huomioon, jotta voidaan luoda realistisia suorituskykytestejä, jotka simuloivat oikeita pelitilanteita oikeissa peliympäristöissä.

Testiajojen automatisoinnista voisi tehdä kokonaan oman insinöörityön omana aiheenaan. Sen toteutukseen liittyy omat haasteensa, jotka ovat valitettavasti tämän insinöörityön aiheen ulkopuolella.

Opinnäytetyössä tehty tapaustutkimus työprojektin suorituskykytestauksesta onnistui. Pelin suorituskykytestit parantuivat huimasti insinöörityön toteutuksen aikana. Lisäksi iOS-laitteiden testausautomaatio on melkein valmis. Työ suorituskykytestauksen parissa jatkuu edelleen, ja kaikkia sen osa-alueita parannellaan pelin koko kehityksen ajan.

Pelien suorituskykytestauksesta Unreal Enginessä on kirjoitushetkellä hyvin vähän julkista tietoa etenkin mobiilipelien puolella, joten tämän opinnäytetyön opeista ja tekniikoista on varmasti apua myös muille aiheesta kiinnostuneille.

Lähteet

1 Howell, Chris. 2022. Modern Game Testing. Reading: Modern Game Testing Company.

2 Murtaza, Touseef. 2020. Testing: Unit VS Integration VS Regression VS Acceptance. Verkkoaineisto. Medium. <https://medium.com/@touseefmurtaza1993/testing-unit-vs-integration-vs-regression-vs-acceptance-a3e190cc54dd>. Päivitetty 18.3.2023. Luettu 27.4.2023.

3 Hoberg, Johan. 2014. Differences between Software Testing and Game Testing. Verkkoaineisto. Game Developer. <https://www.gamedeveloper.com/programming/differences-between-software-testing-and-game-testing>. Päivitetty 21.7.2014. Luettu 27.4.2023.

4 Molyneaux, Ian. 2014. The Art of Application Performance Testing. 2nd Edition. E-kirja. O'Reilly Media.

5 Kinsbruner, Eran. 2022. Automated testing vs manual testing vs continuous testing. Verkkoaineisto. Perfecto. <https://www.perfecto.io/blog/automated-testing-vs-manual-testing-vs-continuous-testing>. Päivitetty 13.12.2022. Luettu 2.4.2023.

6 Axelrod, Arnon. 2018. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. E-kirja. Apress.

7 Grove, Ricky. 2019. Focus: Unreal Engine – A Brief History of Unreal. Verkkoaineisto. Renderosity Magazine. <https://magazine.renderosity.com/article/5330/focus-unreal-engine-a-brief-history-of-unreal>. Päivitetty 23.7.2019. Luettu 2.4.2023.

8 Unreal Engine 5 is now available. 2022. Verkkoaineisto. Unreal Engine. Epic Games. <https://www.unrealengine.com/en-US/blog/unreal-engine-5-is-now-available>. Päivitetty 5.4.2022. Luettu 2.4.2023.

9 Chiatti, Francesco. 2021. Unit-testing with Unreal Engine 4. Verkkoaineisto. Zuru. <https://zuru.tech/blog/unit-testing-with-unreal-engine-4>. Päivitetty 12.2.2021. Luettu 2.4.2023.

10 Command. Verkkoaineisto. Refactoring Guru. <https://refactoring.guru/design-patterns/command>. Luettu 2.4.2023.

11 CSVToSVG Tool. Verkkoaineisto. Unreal Engine. Epic Games. <https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/CSVToSVG/>. Luettu 9.4.2023.

12 Rehkopf, Max. What is continuous integration? Verkkoaineisto. Atlassian. <https://www.atlassian.com/continuous-delivery/continuous-integration>. Luettu 2.4.2023.

13 Kumar, Rajesh. 2022. What is TeamCity and How it works? An Overview and Its Use Cases. Verkkoaineisto. DevOps School. <https://www.devopsschool.com/blog/what-is-teamcity-and-how-it-works-an-overview-and-its-use-cases/>. Päivitetty 6.4.2022. Luettu 29.4.2023.

14 AWS Device Farm. Verkkoaineisto. AWS. <https://aws.amazon.com/device-farm/>. Luettu 29.4.2023.

15 Introduction To Appium: What Is Appium And Its Architecture. 2023. Verkkoaineisto. Software Testing Help. <https://www.softwaretestinghelp.com/what-is-appium/>. Päivitetty 19.3.2023. Luettu 29.4.2023.

16 Thorpe, Dave. Mobile Device Fragmentation – What it is and why should you care. Verkkoaineisto. Indiespring. <https://indiespring.com/appinsights/mobile-device-fragmentation-what-it-is-and-why-should-you-care/>. Luettu 2.4.2023.

17 How do you avoid common pitfalls and challenges in game testing? Let’s find out! Verkkoaineisto. TGG. <https://thegg.net/articles/how-do-you-avoid-common-pitfalls-and-challenges-in-game-testing-lets-find-out/>. Luettu 2.4.2023.

Liitteet

Liite 1: ACameraSpline.h

```
UPROPERTY(EditAnywhere)
bool IncludedInTests;

UPROPERTY(EditAnywhere)
USplineComponent* SplineComponent;

UPROPERTY(EditAnywhere)
int32 OrderNumber;

UPROPERTY(EditAnywhere)
bool UseCustomSpeed;

UPROPERTY(EditAnywhere)
float CustomSpeed;

UPROPERTY(EditAnywhere, Category = StaticShot)
bool IsStaticShot;

UPROPERTY(EditAnywhere, Category = StaticShot)
float StaticShotDuration;

float DefaultSpeed;
float CurrentTimeAtSpline;
float TimeToRunAtSpline;

ACameraSpline::ACameraSpline()
{
     PrimaryActorTick.bCanEverTick = false;
     SplineComponent =
          CreateDefaultSubobject<USplineComponent>(TEXT("Path"));
     RootComponent = SplineComponent;
     IncludedInTests = true;
     DefaultSpeed = 700.0f;
     CustomSpeed = 500.0f;
}

void ACameraSpline::BeginPlay()
{
     Super::BeginPlay();

     if (IsStaticShot)
     {
          TimeToRunAtSpline = StaticShotDuration;
          return;
     }

     TimeToRunAtSpline = UseCustomSpeed
          ? CalculateTimeToRunAtSpline(CustomSpeed)
          : CalculateTimeToRunAtSpline(DefaultSpeed);
}
```

```
float ACameraSpline::CalculateTimeToRunAtSpline(
     const float SpeedModifier) const
{
     const auto SplineLength = SplineComponent->GetSplineLength();
     const auto CalculatedTime = SplineLength / SpeedModifier;
     return CalculatedTime;
}

void ACameraSpline::Tick(const float DeltaSeconds)
{
     CurrentTimeAtSpline += DeltaSeconds;
     CurrentTimeAtSpline = FMath::Clamp(CurrentTimeAtSpline,0.0f,Time-
ToRunAtSpline);

     const int CurrentPercentage =
          FGenericPlatformMath::RoundToInt(
               CurrentTimeAtSpline / TimeToRunAtSpline * 100);
     GEngine->AddOnScreenDebugMessage(-1,
               GWorld->GetDeltaSeconds(),
               FColor::Green, FString::Printf(
                    TEXT("Path progress: %i%%"), CurrentPercentage));
}

bool ACameraSpline::HasTimeEnded() const
{
     return CurrentTimeAtSpline >= TimeToRunAtSpline;
}

bool ACameraSpline::HasReachedEnd() const
{
     return CalculateNormalizedSplinePosition() == 1.0f;
}


float ACameraSpline::CalculateNormalizedSplinePosition() const
{
     const float NormalizedTime = FMath::Clamp(
          CurrentTimeAtSpline/TimeToRunAtSpline,0.0f,1.0f);

     return NormalizedTime;
}

FRotator ACameraSpline::GetCurrentRotationAtTime() const
{
     if(SplineComponent == nullptr)
     {
          return FRotator::ZeroRotator;
     }

     const FRotator Rotator = SplineComponent->GetRotationAtTime(
          CalculateNormalizedSplinePosition(),
          ESplineCoordinateSpace::World, true);
     return Rotator;
}

int32 ACameraSpline::GetOrderNumber() const
{
     return OrderNumber;
}
```

```
bool ACameraSpline::IsIncludedInTests() const
{
     return IncludedInTests;
}

FVector ACameraSpline::GetCurrentPositionAtTime() const
{
     if( SplineComponent == nullptr )
     {
          return FVector::ZeroVector;
     }

     const FVector Location = SplineComponent->GetLocationAtTime(
          CalculateNormalizedSplinePosition(),
          ESplineCoordinateSpace::World, true);
     return Location;
}
```

Liite 2: FPerformanceTestRunner.h

```
virtual void FPerformanceTestRunner::GetTests(
     TArray<FString>& OutBeautifiedNames,
     TArray<FString>& OutTestCommands) const override
{
     OutBeautifiedNames.Add(“MyMap1”);
     OutTestCommands.Add(“MyMap1”);

     OutBeautifiedNames.Add(“MyMap2”);
     OutTestCommands.Add(“MyMap2”);
}

virtual bool FPerformanceTestRunner::RunTest(
     const FString& InParameter) override
{
     FString MapPath = "";
     FString SublevelPath = "";
     bool bAutoStartEncounters;
     InitializeParameters(InParameter, MapPath, SublevelPath);

     IConsoleVariable* CVAR = IConsoleManager::Get()
          .FindConsoleVariable(TEXT("PerformanceRun"));
     CVAR->Set(true);

     if (GEngine->IsEditor())
     {
          ADD_LATENT_AUTOMATION_COMMAND(FRequestSessionStart())
     }

     ADD_LATENT_AUTOMATION_COMMAND(FLoadMapAndWait(MapPath))
     ADD_LATENT_AUTOMATION_COMMAND(StreamSublevelCommand(SublevelPath))
     ADD_LATENT_AUTOMATION_COMMAND(RunPerfCameraAtSplineCommand())

     return true;
}

void FPerformanceTestRunner::InitializeParameters(
     const FString& InParameter, FString& OutLevelPath,
     FString& OutSublevelPath, bool& OutAutoStartEncounters)
{
     if (InParameter.Compare(“MyMap1”) == 0)
     {
          OutLevelPath = PATH_TO_MY_MAP_1;
          OutSublevelPath = PATH_TO_MY_SUBLEVEL_OF_MAP_1;
     }

     if (InParameter.Compare(“MyMap2”) == 0)
     {
          OutLevelPath = PATH_TO_MY_MAP_2;
          OutSublevelPath = PATH_TO_MY_SUBLEVEL_OF_MAP_2;
     }
}
```

Liite 3: FRequestSessionStartCommand.h

```
virtual bool FRequestSessionStartCommand::Update() override
{
     if(!IsRequestStarted)
     {
#if WITH_EDITOR
          /// Jos editorissa, asetetaan arvot sessiolle ja avataan uusi
          /// pelisessio

          FRequestPlaySessionParams SessionParams;
          SessionParams.EditorPlaySettings =
               NewObject<ULevelEditorPlaySettings>();
          SessionParams.EditorPlaySettings->NewWindowWidth = 1280;
          SessionParams.EditorPlaySettings->NewWindowHeight = 720;
          SessionParams.EditorPlaySettings->GameGetsMouseControl =
                                                        false;
          GUnrealEd->RequestPlaySession(SessionParams);
#endif
          IsRequestStarted = true;
     }

     /// Pelisessio on valmiina heti kun maailma löytyy
     return GWorld != nullptr;
}
```

Liite 4: APerformanceCameraController.h

```
void OnActivate(APlayerController* OriginalPlayerController)
{
     /// Kopioidaan kameraohjaimelle sama sijainti ja kiertosuunta kuin
     /// alkuperäisellä kameralla
     FVector OriginalCameraLocation;
     FRotator OriginalCameraRotation;
     OriginalPlayerController->GetPlayerViewPoint(
          OriginalCameraLocation, OriginalCameraRotation);
     float const OriginalCameraFOV =
          OriginalPlayerController->PlayerCameraManager->GetFOVAngle();

     SetInitialLocationAndRotation(OriginalCameraLocation,
          OriginalCameraRotation);

     /// Asetetaan kameralle sama näkökentän arvo kuin alkuperäisellä
     /// kameralla
     if (PlayerCameraManager)
     {
          PlayerCameraManager->SetFOV(OriginalCameraFOV);
          PlayerCameraManager->UpdateCamera(0.0f);
     }

     /// Lisätään tiedot tästä kameraohjaimesta maailmalle
     GetWorld()->AddController(this);
```